AD-A100 915 YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE F/G 9/2
SECONDARY STORAGE METHODS FOR SOLVING SYMMETRIC, POSITIVE DEFIN--ETC(U)
APR 81 J R PERRY N00014-76-C-0277
UNCLASSIFIED RR-201 NL

1 OF 1
AD A
100915

END
DATE
FILMED
8-81
DTIC

LEVEL II

1

LUX ET VERITAS

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

This work was presented to the faculty of the Graduate School of Yale University in candidacy for the degree of doctor of Philosophy.

Secondary Storage Methods for Solving Symmetric, Positive Definite, Banded Linear Systems.

John Richard/Perry

Research Report #201

April 1981

This work was supported in part by ONR Grant N00014-76-C-0277, AFOSR Grant F49620-77-C-0037, and ERDA Grant SRSA-EG-77-S-02-4349

ABSTRACT

Secondary Storage Methods for Solving Symmetric, Positive Definite, Banded Linear Systems

John Richard Perry

Yale University, 1981

The solution of a linear system of equations

$$Ax=b$$

is often computed using variations of Gaussian elimination. A major problem with these methods is the *large storage requirement* to compute the factorization of matrices that arise in practice. We focus on the Cholesky method for factoring banded, symmetric, positive definite matrices, which arise in finite-difference and finite-element simulations. We develop and analyze methods for using secondary storage in cases where there is not enough primary memory to compute the band Cholesky factorization.

First, we study the performance of virtual memory paging systems, which automatically perform the disk I/O necessary to execute programs that do not fit into primary memory. We survey results for Gaussian elimination in general and identify techniques for reducing the paging costs of the band Cholesky algorithm in particular. We show that paging does not result in a smooth tradeoff between primary memory usage and I/O costs for this problem.

We present secondary storage *methods that explicitly* deal with the use of memory and I/O in computing the Cholesky factorization. The methods are based on partitionings of the band elements into strips, containing columns of the band, or blocks, containing submatrices. For a bandwidth of M, the methods use from $O(M^2)$ to $O(1)$ primary memory. We analyze the costs for each method and show that secondary storage methods offer a more efficient tradeoff between memory usage and I/O costs than does paging. Further, the memory occupancy costs of these methods are shown to be as low, asymptotically, as those of any direct or iterative method for solving *linear systems* in primary memory.

Many computer configurations used for scientific computation allow for control over the parallel execution of I/O and computation. For several of the secondary storage methods, we present storage and I/O schemes that allow *I/O* to be *overlapped* with computation. We derive conditions under which the factorization is compute-bound, i.e., all I/O between the initial input and the final output is completely hidden behind concurrent computation. Further, we show that the amount of memory needed to achieve compute-boundedness is independent of the size and bandwidth of the matrix. Thus, for a given processor speed and I/O rate, a constant amount of memory is sufficient to keep the processor busy during the band Cholesky factorization.

Finally, we present the results of implementing these methods on a DEC-System 2060 using secondary disk storage. Timing experiments show that the use of primary memory can be reduced to N^2 with almost no increase in turn-around time, and that computation dominates I/O even at the lowest levels of primary memory usage.

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Definition of the Problem

In this dissertation, we are concerned with the problem of solving a linear system of equations,

$$Ax = b.$$

This is one of the most thoroughly-studied problems in numerical computation. Yet, because of the variety of applications in which linear systems arise, the large proportion of computer time spent solving them, and the evolution of machine architectures, it continues to be an active area of research.

In particular, we shall investigate the storage aspects of variants of Gaussian elimination for solving linear systems. Gaussian elimination is generally considered to be a good method because it computes an exact solution, with certain bounds on *round-off error*, in a specific number of steps. In practice, a major problem is that the storage requirement of Gaussian elimination can be quite large. Iterative methods [40] are often used because they require less storage.

Much research has been directed at reducing the storage requirements of Gaussian elimination (and usually the work involved as well) by exploiting the symmetry and/or the zero structure of the coefficient matrix. The easiest such structure to exploit is that of a band matrix [8, 22]. By "easy", we mean that there is little additional overhead due to reordering of equations and unknowns or manipulation of the data structures necessary to store and operate upon the matrix. In some problems, more arithmetic operations and storage can be avoided by using a profile storage scheme [18]. Profile methods are sometimes referred to as envelope or skyline methods. In some problems, one can do still better by using general sparse algorithms, which store and operate upon only the nonzero elements [12, 34]. However, the work and storage advantages of profile and sparse methods are offset by higher overhead costs. Furthermore, there still may be a storage problem with any of these approaches in the sense that the amount of memory limits the size of a system that can be stored and solved on a given machine.

A general approach to solving problems which require more storage than is available in main memory is to use some form of backup storage. In most computing environments, a limited amount of fast <u>primary</u> memory is backed up by a slower but much larger <u>secondary</u> memory, the most common being disk storage. Often, an automatic mechanism for using secondary storage is provided by a virtual memory operating system. Research on the use of such operating systems for matrix computations aims at reducing the paging costs through programming

techniques [23, 24, 32, 37, 14] or compiler design [1]. We shall summarize some of these approaches and show ways to organize band Gaussian elimination so as to minimize the *number of page faults*.

However, paging systems are not available on many of the machines used for large matrix computations. Indeed, machine architectures can have memory configurations and transfer mechanisms that are too complex or require too much control for a paging approach to be useful. Moreover, an innate drawback of automatic *paging systems is that for any* particular algorithm, they cannot perform as well as an explicit *individually-tailored I/O* scheme.

The principal aim of this dissertation is to develop and analyze __secondary storage methods__, which incorporate a strategy for the explicit transfer of data between primary and secondary storage as part of the algorithms for solving linear systems. We shall focus on algorithms for solving symmetric, positive *definite*, banded *linear* systems. This type of system arises in many applications, especially finite-difference and finite-element methods for elliptic partial differential equations. In the next section we introduce one such example to serve as a model problem and discuss the reasons for using band elimination, as opposed to profile or sparse elimination, for developing secondary storage methods.

Some work exists on the implementation of Gaussian elimination using secondary storage. Several codes for solving symmetric, positive

definite banded systems have been developed, *especially as part of* structural analysis packages [15, 25, 30, 33, 39]. In particular, the frontal method [17] uses secondary storage as it concurrently generates and solves the linear systems arising from finite elements. Similar issues have been studied for specific machines such as the Cray-1 [3, 20, 26].

We extend these efforts by defining a class of secondary storage methods for banded systems, a few of them similar to those in the cited works. These methods allow Gaussian elimination algorithms to run within reduced (in some cases arbitrarily small) amounts of primary memory at the cost of performing the necessary I/O. Thus, secondary storage capacity becomes the only limiting factor on the size of a system that can be solved on a given machine. Furthermore, our methods *differ from most* by allowing the size of records involved in transfers *to be specified* by the user. Through this mechanism, each method offers a tradeoff within limits between the amount of primary memory used and the amount of *I/O*. Using a simple *model for the cost of performing I/O*, we analyze and compare the I/O and memory occupancy costs for the various methods. This analysis reveals that the tradeoff between memory usage and I/O can be better exploited with secondary storage methods than with paging. Also, since it is possible with certain architectures to carry out I/O concurrent with computation, we consider the possibilities and implications of overlapping the I/O with the arithmetic work of these secondary storage methods.

In the remaining sections of this chapter, we introduce terminology and notation and outline the organization and results of the dissertation.

1.2 The Model Problem

In order to compare the costs of secondary storage methods with other approaches to reduce primary storage, we now present a specific model problem: the solution of Poisson's equation,

$$U_{xx}+U_{yy} = F(x,y),$$

on a bounded square domain, where the values of U on the boundary are known. If a five-point difference operator is applied over a regular M by M grid, a symmetric, positive definite linear system results in which $N=M^2$ [35]. The coefficient matrix has only two nonzero off-diagonal elements above the diagonal in each row.

The form that this sparsity takes, and the resulting work and storage requirements, depends on the ordering of the equations and unknowns. In Figure 1-1 we show the matrix with bandwidth M produced by row ordering and the profile matrix produced by diagonal ordering. This profile matrix requires less work and storage to factor than the band matrix, since the profile is a subset of the band. Further reductions of work and storage are realized by ordering the equations and unknowns by the method of nested dissection [12] and using sparse elimination to solve the resulting system.


```
      B    B    B    B        | X X 0 0 X                         |
      |    |    |    |        |   X X 0 0 X                       |
B---1----2----3----4---B      |     X X 0 0 X           0         |
      |    |    |    |        |       X 0 0 0 X                   |
B---5----6----7----8---B      |         X X 0 0 X                 |
      |    |    |    |        |           X X 0 0 X               |
B---9---10---11---12---B      |             X X 0 0 X             |
      |    |    |    |        |               X 0 0 0 X           |
B--13---14---15---16---B      |                 X X 0 0 X         |
      |    |    |    |        |                   X X 0 0 X       |
      B    B    B    B        |                     X X 0 0 X     |
                              |                       X 0 0 0 X   |
                              |                         X X 0 0   |
                              |                           X X 0   |
                              |                             X X   |
                              |                               X   |
```

Natural Ordering ====> Symmetric Banded Matrix

```
      B    B    B    B        | X X X                             |
      |    |    |    |        |   X 0 X X                         |
B---1----2----4----7---B      |     X 0 X X                       |
      |    |    |    |        |       X 0 0 X X                   |
B---3----5----8---11---B      |         X 0 0 X X                 |
      |    |    |    |        |           X 0 0 X X               |
B---6----9---12---14---B      |             X 0 0 0 X             |
      |    |    |    |        |               X 0 0 X X           |
B--10---13---15---16---B      |                 X 0 0 X X         |
      |    |    |    |        |                   X 0 0 X         |
      B    B    B    B        |                     X 0 0 X       |
                              |                       X 0 X X     |
                              |                         X 0 X     |
                              |                           X 0 X   |
                              |                             X X   |
                              |                               X   |
```

Diagonal Ordering ====> Symmetric Profile Matrix

Figure 1-1: Linear Systems Arising from the Model Problem

We focus on band methods largely because the theoretical reductions offered by profile and sparse methods are gained through the use of more complex data structures. The storage of band elements is accomplished using a standard rectangular matrix in which the location of any element can be directly determined from its indices and from N or M. Profile and sparse storage schemes use pointers, which means that addresses are indirectly determined from the data stored in one or more index arrays. This introduces additional work and storage overhead.

Secondary storage methods are particularly useful for the machines known as peripheral array processors [2]. These are inexpensive machines that offer the speed of much larger mainframes, but have limited memory space backed up by the memory of the host machine, or by direct access to disk or bulk storage. However, they have architectures that are most effective with certain kinds of algorithms. In particular, they have a limited capacity for the type of indirect and data-dependent addressing that sparse algorithms require. Therefore, we concentrate on band elimination, which has an elementary data structure and algorithms that are well-suited for parallel organization.

1.3 Organization and Preview of Results

In Chapter 2, we survey the Gaussian-elimination-type algorithms for solving linear systems. The method of primary interest is the Cholesky factorization for symmetric, positive definite, banded systems,

which can be stored and ordered either by columns or by blocks. We discuss the issue of locality, which affects the paging performance of the algorithms and the amounts of memory and I/O required by secondary storage methods.

In Chapter 3, we summarize some of the published results on solving linear systems with paging. We add some observations applying to banded systems, and point out situations in which paging is inefficient in its levels of I/O or primary memory usage. In particular, we find that for each algorithm that is organized for paging efficiency, there is a threshold in the amount of primary memory above which there is a minimal amount of paging, and below which there is an order of magnitude more paging.

In Chapter 4, we present secondary storage methods for solving symmetric, positive definite banded systems. A method is defined by a partitioning of the coefficient matrix into records containing either strips or blocks from the band, and a strategy for coordinating work, storage and I/O while computing the factorization. The strip or block size is variable, allowing a certain degree of control over the storage and I/O demands of each method.

The various costs associated with secondary storage methods are quantified in Chapter 5. We characterize I/O costs in terms of a simple linear model of transfer time which takes into account the number of I/O events as well as the number of elements transferred. We use this model

to derive expressions for the amount of I/O in each secondary storage method, and to quantify memory occupancy costs. For a symmetric banded system of dimension N and bandwidth M, the methods span the range from $O(M^2)$ to $O(1)$ primary memory, as opposed to NM memory required to store the entire band. As the amount of required primary storage decreases through this range, the I/O costs rise from $O(N)$ events of $O(M)$ elements each up to $O(NM^2)$ events of $O(1)$ elements each. This analysis identifies some of the factors affecting the best choice of method and parameters given the size of the problem, the amount of memory, and the characteristics of I/O costs.

In Chapter 6 we examine the implications and capabilities of parallel execution of I/O and computation events. For methods using $O(M^2)$ memory, we introduce schemes for allocating memory between computation and buffering, and for overlapping I/O with computation to *minimize the time that the processor is idle and* thus the turn-around time. An analysis of these schemes shows the amount of primary memory that is needed to overlap nearly all I/O during the factorization. Finally, we show that nearly all I/O can be overlapped even in the secondary storage factorization method with the highest level of I/O. This result implies that *we can compute the Cholesky factorization* within a constant amount of primary memory (i.e., **independent** of N and M) with virtually no increase in time due to I/O.

A more general implication of this result is that the time and

space requirements of an algorithm are not the only criteria affecting the cost of computation. The capability for a controlled flow of data can replace the storage requirements of certain types of algorithm. Memory hierarchies occur in many computing environments, and to use them effectively requires careful study of the data flow characteristics of the algorithms that are most common in numerical computing.

Finally, in Chapter 7, we describe the features of a package that implements the methods of Chapter 4 called BESS, for Band Elimination with Secondary Storage. We report on the performance of these codes *using a DEC-System 2060 with secondary disk storage.*

1.4 Notation and Terminology

To complete this introduction, we present the simplest form of *Gaussian elimination*, Algorithm 1-1, in order to introduce notation for specifying algorithms. We define a <u>step</u> of such an algorithm as being one execution of the outermost loop. In the forward elimination stage of the algorithm (Lines 1-7), each step introduces zeroes in the j^{th} column below the diagonal so that, after N-1 steps, A is *upper-triangular*.

We shall represent multiplication by juxtaposition, as with "$S\ b_i$" and "$A_{ji}\ x_i$" in Lines 5 and 11 of Algorithm 1-1, respectively. We use the equals sign for both assignment and comparison testing, as the

```
1.  FOR j = 1 TO N-1 DO
2.    [ P = 1/A_jj ;
3.      FOR i = j+1 TO N DO
4.        [ S = P A_ij ;
5.          b_i = b_i - S b_j ;
6.          FOR k = j TO N DO
7.            [ A_ik = A_ik - S A_jk ] ] ] ;
8.  FOR j = N TO 1 STEP -1 DO
9.    [ x_j = b_j/A_jj ;
10.     FOR i = 1 TO j-1 DO
11.       [ b_i = b_i - A_ij x_j ] ]
```

Algorithm 1-1: *Gaussian Elimination* (no Pivoting)

---

context indicates. The index of a "FOR...DO" loop, delimited by brackets and indenting, is incremented by 1 unless "STEP -1" is specified to indicate decrementing as in Line 8 of Algorithm 1-1. In cases where the limits of a loop encompass no values (such as when $j=1$ in Line 10 of Algorithm 1-1) the loop is to be ignored. Finally, the indexing of elements of a matrix within the algorithms will always refer to their positions in the full matrix, regardless of the actual storage scheme. This convention is aimed at maintaining as much consistency as possible between various forms of the algorithm that require different

storage schemes (nonsymmetric, symmetric, band, etc.). The problem of mapping these indices into the actual location of the elements in memory is left as a detail of implementation.

Also note that, although we refer to elements of b and x, the algorithm is ordered so that x can overwrite b as it is computed. Since we are concerned with conserving storage, this convention will be maintained for all vectors and all matrices involved in each algorithm of this dissertation.

We now introduce terminology (similar to that of Mondkar and Powell [25]) to help describe the locality, or pattern of references to matrix elements, within algorithms. The algorithms we are considering perform inner or outer products within nested loops, of the general form

```
FOR i = . . .
  FOR j = . . .
    FOR k = . . .
      References to ( A_ij, A_ik, A_jk ).
```

The first element of the triple is referenced the most locally in that its indices are independent of the innermost loop index. We call this the <u>principal element</u> during the computation with that triple. The indexing in algorithms throughout this dissertation is such that A_{ij} is the principal element within the innermost loop. For instance, within the outer product in Line 7 of Algorithm 1-1, S (as computed from A_{ij}) is the principal element. The second element is the next-most locally referenced element, or that without the next-innermost loop index (A_{ik} in this example). We define this element to be the <u>first subordinate</u>

element and the remaining, least-locally referenced element to be the second subordinate element.

Thus, each principal element is associated with sets of *first* and second subordinate elements which we respectively call its first and second subordinate sets. Further, the successive principal elements in a specified portion of an algorithm can be referred to as a principal set, whose first and second subordinate sets are the unions of the subordinate sets of its elements. In Figure 1-2, we illustrate the subordinate sets corresponding to a principal element and a principal column in the Gaussian Elimination algorithm. Finally, notice from Algorithm 1-1 and Figure 1-2 that all elements of the second subordinate are modified in each step of the outer-product algorithms. We shall see that such characteristics affect the amount of paging or I/O in factorization algorithms.

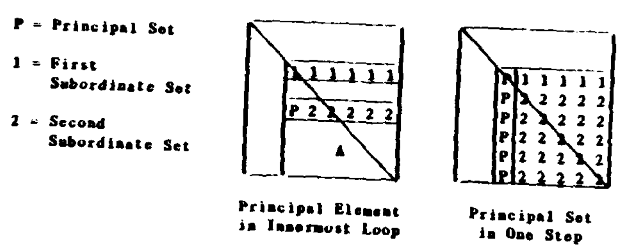


Figure 1-2: Principal and Subordinate Sets in Gaussian Elimination

CHAPTER 2

Gaussian Elimination and Its Variations

2.1 Factorization and Substitution

There is a variety of algorithms for solving linear systems by factorization. These algorithms have been extensively studied and their properties with respect to accuracy and efficiency are well understood. Wilkinson [38] and Forsythe and Moler [8] contain good overviews. The variety exists in order to efficiently solve the many special cases that arise in practice. Some algorithms can reduce the work and storage required to solve the system by exploiting properties of the coefficient matrix. Other algorithms reorder the operations to improve the locality and thereby reduce the amount of paging involved when the algorithms are executed in a virtual memory environment. In this chapter we introduce these algorithms and identify the characteristics that are significant for secondary storage methods.

The method of Gaussian elimination (Algorithm 1-1) for solving a linear system

$$Ax = b$$

of dimension N is commonly expressed in the equivalent form of computing the unique LU factorization of A, where L is unit-lower-triangular and U is upper-triangular. Once L and U are found, the solution for one or more right-hand sides is obtained by solving the triangular systems

$$Ly = b \text{ and } Ux = y$$

by forward and backward substitution, respectively. We refer to the computation of L and U as the factorization stage and the solution of the triangular systems as the forward-solve and back-solve stages of the algorithm, respectively. Our primary concern is the factorization stage, since it requires an order of magnitude more work than the solution of the triangular systems.

In Section 2, we present inner- and outer-product algorithms for computing L and U when A is nonsymmetric and dense (i.e., no elements are assumed to be zero). In Section 3, we present the Cholesky factorization for symmetric, positive definite A, which can cut the work and storage requirements in half. In Section 4, we describe other variations of these algorithms which exploit the zero structure of a band or profile coefficient matrix to gain further savings in work and storage. This thesis focuses on secondary storage methods for the symmetric positive definite banded case because the locality and structure of the algorithm allows large savings in primary storage with low levels of I/O. The general approach of some of the methods extends to other matrix structures, which we shall point out where appropriate. Finally, in Section 5, we describe block factorization methods. Block

methods have better locality than standard row- or column-oriented algorithms [23] and hence decrease the ratio of I/O to work within a given amount of primary memory.

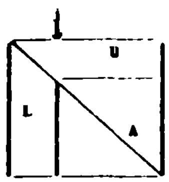
In conjunction with the algorithms, we shall specify storage schemes that satisfy several efficiency and convenience considerations. In particular, it is advantageous for the vector operations of the innermost loops of the algorithms to be performed on vectors that are contiguous in memory. Among the advantages are

- it simplifies the coding of the algorithm;
- the code generated by many compilers will run faster;
- the convenience or efficiency can be even greater when the vector operations are performed by optimized vector subroutines (such as the BLAS in LINPACK [5]), assembly-language routines, or by vector processors;
- memory references tend to be local and thus induce less paging, as first observed by Moler [24].

Furthermore, I/O operations are generally easier and faster if they transfer to or from contiguous areas of memory.

2.2 Outer- and Inner-Product Forms of Dense LU Factorization

If an LU factorization of A exists (see [8, 38]), there are several algorithms for computing the elements of L and U. The algorithms are algebraically equivalent, but differ in the order in which operations *are carried out. We focus on two such orderings.*



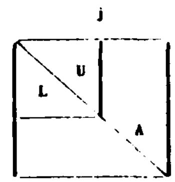

```
1. FOR j = 1 TO N DO
2.   [ FOR i = j TO N DO [ U_ji = A_ji ] ;
3.     FOR i = j+1 TO N DO
4.       [ L_ij = A_ij/U_jj ;
5.         FOR k = j+1 TO N DO
6.           [ A_ik = A_ik - L_ij U_jk ] ] ]
```

Algorithm 2-1: Outer-Product Dense LU Factorization

In one case the algorithm performs vector <u>outer-products</u> (that is, the innermost loop is o. the form V2 = V2 - S*V1 where S is a scalar, invariant in the loop, and V1 and V2 are vectors) and in the other case <u>inner-products</u> (of the form S = S - V1*V2). Each requires about $N^3/3$ multiplies and N^2 storage.

The <u>outer-product</u> form of LU factorization, Algorithm 2-1, is closest to Gaussian elimination. This algorithm reduces A to upper-triangular form one column at a time by adding multiples of the i^{th} row to the remaining (N-i) by (N-i) submatrix so as to create zeroes below the diagonal in the i^{th} column. U then contains the upper-triangular result and L is composed of the multipliers used to eliminate elements of the lower triangle.

The other algorithm of interest is the <u>inner-product</u> form, Algorithm 2-2, also known as the Crout method. The innermost loops of




```
1. FOR j = 1 TO N DO
2.   [ FOR i = 1 TO j-1 DO
3.     [ FOR k = 1 TO i-1 DO
4.         [ A_ji = A_ji - L_jk U_ki ] ;
5.       L_ji = A_ji/U_ii ] ;
6.     FOR i = 1 TO j DO
7.     [ FOR k = 1 TO i-1 DO
8.         [ A_ij = A_ij - L_ik U_kj ] ;
9.     U_ij = A_ij ] ]
```

Algorithm 2-2: Inner-Product Dense LU Factorization

---

this algorithm carry out inner products between elements from a row of L and a column of U. In essence, this is a reordering of the factorization so that all modifications to a given element are made in succession.

There are, in fact a total of six distinct algorithms for computing the LU factorization, corresponding to the six possible permutations of the indices in the expression

$$A_{ij} = A_{ij} - L_{ik}U_{kj}$$

For example, the inner-product algorithm computes the elements by *rows* rather than columns of U if we exchange i and j in Lines 1, 2, 3, 6, and 7 of Algorithm 2-2. Since the operations that modify a given element

are carried out in the same order, the round-off properties of these algorithms are generally similar. However, in the inner-product algorithm, the inner products can be accumulated in a double-precision register to reduce round-off error [8] while using virtually no extra storage.

For a general matrix A, it may be necessary to perform some type of pivoting during *factorization. Otherwise, division by a zero or* near-zero element on the diagonal may making the algorithm either ill-defined or unstable. Complete pivoting guarantees stability, but partial pivoting is generally sufficient in practice [38]. However, in cases where A is symmetric and positive definite (as is true in many applications) it can be shown that no pivoting is necessary [8]. We shall concentrate on symmetric, positive definite banded linear systems and shall therefore not discuss pivoting in any detail, except to mention what methods do or do not easily extend to include pivoting.

We show several *examples of principal sets and their subordinate* sets in Figure 2-1 to illustrate the pattern of memory references that occur in these two factorization algorithms. The inner-product form has several advantages over the outer-product form, especially in a secondary storage context. The main advantage is that the modification or rewriting of elements is more local. For each principal row and column, the inner-product algorithm reads but does not modify the second subordinate set, in the upper left j by j submatrix. The outer-product

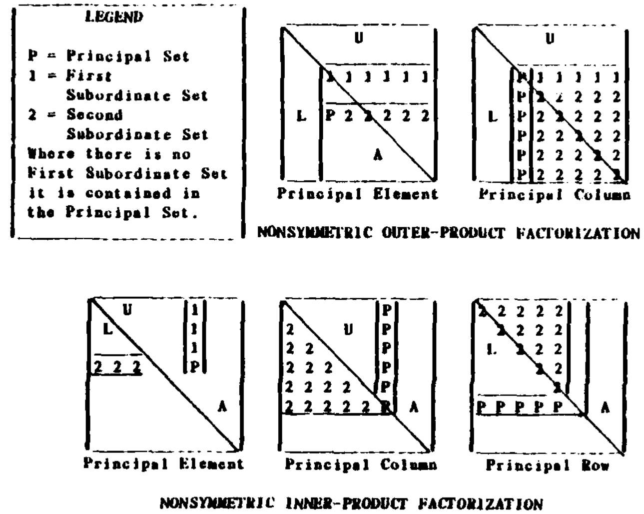


Figure 2-1: Principal and Subordinate Set Examples

form rewrites its second subordinate set, the lower right right N-j by N-j submatrix, which would involve additional I/O if these elements were in secondary storage. Even when the entire matrix is in primary storage and no I/O is involved, the memory rewrite costs can cause the outer-product form to be less efficient.

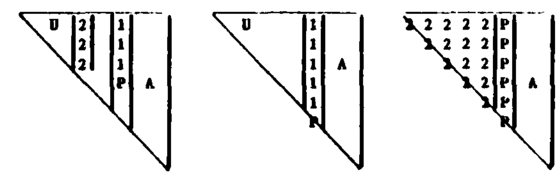
2.3 The Cholesky Method for Symmetric, Positive Definite Systems

If A is symmetric and positive definite, we can halve the work and storage by using the Cholesky method to compute the U^TU factorization. Algorithm 2-3 is the inner-product form of the Cholesky factorization. In addition, Figure 2-2 shows examples of its principal and subordinate sets, using the notation of Figure 2-1.

We should also mention that there is a variation of the Cholesky algorithm that factors A into U^TDU, where U is unit-upper-triangular and D is diagonal. This factorization does not require any square roots, and thus is computationally more efficient. However, in the context of this dissertation, we prefer the square-root method because it is more local in its memory references. In particular, in the loop corresponding to Line 5 of Algorithm 2-3, the square-root-free method references previous diagonal elements A_{kk} in addition to all elements of the j^{th} column. In some secondary storage schemes, this would induce additional I/O or require a more complicated storage scheme, which would offset the computational advantage.

1. FOR $j = 1$ TO N DO
2. [FOR $i = 1$ TO $j-1$ DO
3. [FOR $k = 1$ TO $i-1$ DO [$A_{ij} = A_{ij} - U_{ki}U_{kj}$] ;
4. $U_{ij} = A_{ij}/U_{ii}$] ;
5. FOR $k = 1$ TO $j-1$ DO [$A_{jj} = A_{jj} - U_{kj}^2$] ;
6. $U_{jj} = (A_{jj})^{1/2}$]

Algorithm 2-3: Inner-Product Cholesky Factorization ($A=U^TU$)



Principal Element — Principal Diagonal Element — Principal Column

Figure 2-2: Principal and Subordinate Sets in the Cholesky Algorithm

Algorithm 2-4 is the forward-back-solve for the Cholesky factorization. In Lines 1-4, the lower-triangular system $U^Ty=b$ is solved, and in Lines 5-8, the upper-triangular system $Ux=y$ is solved. The forward-solve is ordered so that the elements of U are used in the same order as they are computed in Algorithm 2-3. Thus, the

```
1.  FOR j = 1 TO N DO
2.    [ FOR i - 1 TO j-1 DO
3.        [ b_j = b_j - U_ij y_i ] ;
4.      y_j = b_j/U_jj ] ;

5.  FOR j = N TO 1 STEP -1 DO
6.    [ x_j = y_j U_jj ]
7.      FOR i - 1 TO j-1 DO
8.        [ y_i = y_i - U_ij x_j ] ]
```

Algorithm 2-4: Forward-Back-Solve for the Cholesky Factorization

forward-solve can be carried out along with the factorization, as it is in Gaussian Elimination. This is usually done in secondary storage *methods to avoid redundant I/O.*

2.4 The Cholesky Factorization of Banded Matrices

The linear systems arising from many applications are sparse, that is, most elements of the coefficient matrix are zero. The work and storage requirements of factorization algorithms can be greatly reduced by exploiting the zero structure. However, zero elements can "fill in" (become nonzero) during the factorization, changing the zero structure of A. A sparse form that is unchanged by fill-in during factorization is that of a symmetric, positive definite, banded matrix [22].

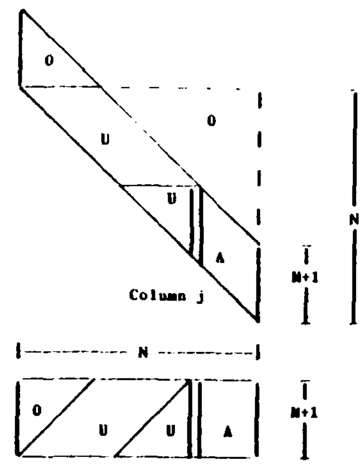


Figure 2-3: Column Storage of the Band of A

We say that a symmetric matrix A is a <u>band matrix of bandwidth M</u> if $A_{ij}=0$ for all $j-i>M$. Algorithm 2-5 is the Cholesky algorithm modified to avoid unnecessary operations outside the band by a simple change in the range of the loop indices. Combined with a symmetric band storage scheme, this reduces the storage from $N^2/2$ to $N(M+1)$ and the number of multiplies from $N^3/6$ to $NM^2/2$, a substantial savings if $M<<N$.

Figure 2-3 shows a mapping of the elements of the upper band of A into dense storage by columns within an N by M+1 matrix. The scheme of

FOR $j = 1$ TO N DO

  [FOR $i = \mathrm{MAX}(1, j-M)$ TO $j-1$ DO

    [FOR $k = \mathrm{MAX}(1, j-M)$ TO $i-1$ DO [$A_{ij} = A_{ij} - U_{ki}U_{kj}$] ;

      $U_{ij} = A_{ij}/U_{ii}$] ;

  FOR $k = \mathrm{MAX}(1, j-M)$ TO $j-1$ DO [$A_{jj} = A_{jj} - U_{kj}^2$] ;

  $U_{jj} = A_{jj}^{1/2}$]

Algorithm 2-5: Inner-Product Band Cholesky Factorization

---

storing band elements by columns is used in LINPACK [5] but is not universal. The IMSL format for storing band matrices [16] is to store the $M+1$ diagonals of the band within the columns of an N by $M+1$ matrix. This is inefficient with respect to paging because a given row or column of the band is spread across nearly the entire extent of memory being used. For the reasons mentioned in the introduction to this chapter, we choose a storage scheme to keep the memory references of an algorithm as local as possible.

2.5 Block Factorization Algorithms

All the factorization algorithms presented so far are <u>scalar</u> algorithms, since the operations being carried out on the coefficient matrix are with scalar arguments. A property of scalar algorithms is that each arithmetic operator has a fixed ratio between the time

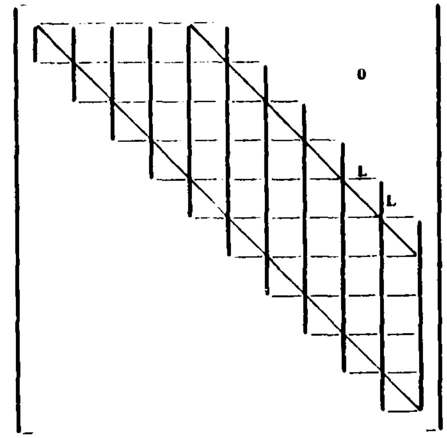


Figure 2-4: Block Partitioning of Symmetric Band Matrix, $\bar{N}=10$, $\bar{M}=4$

---

required for the computation and the time required to transfer the arguments to and/or *from* memory. The value of such a computation-to-I/O ratio for each operator is determined by the relative speeds of the processor and the memory device holding the arguments.

In this section, we present a generalization of the scalar band Cholesky factorization to a <u>block</u> algorithm, that is, one in which the primitive operators are matrix operations carried out on square submatrices of the coefficient matrix. In Figure 2-4, we show an example of a symmetric band matrix partitioned into blocks of dimension

Scalar Operator	Corresponding Block Operator	
	Operator Notation	Operator Definition
$a - bc$	$\{A\} - \{B\}^T\{C\}$	Algorithm 2-6
a/d	$\{A\}/\{D\}$	Algorithm 2-7
$d^{1/2}$	$\{D\}^{1/2}$	Algorithm 2-3
In back-solve: x/d	$\{x\}\backslash\{D\}$	Algorithm 2-8

Table 2-1: Block Operators for Inner-Product Cholesky Algorithm

L, where we define $\bar{N}=\lceil N/L\rceil$ and the <u>block bandwidth</u> $\bar{M}=\lceil M/L\rceil$.

We can convert the scalar algorithm into a block algorithm by reordering the operations to maximize locality within this block structure. This is equivalent to generalizing the operators and arguments to refer to blocks rather than individual elements. We use the notation $\{A\}_{ij}$, $1 \le i,j \le \bar{N}$, to denote the appropriate block of A. The right-hand-side vector is also partitioned for the purposes of operating with these blocks. Thus, $\{b\}_j$ denotes elements $\{b_i \mid (j-1)L < i \le jL\}$.


```
1.  FOR j = 1 TO L DO*
2.    [ FOR i = 1 TO L DO
3.        [ FOR k = 1 TO L DO  [ A_ij = A_ij - B_ki C_kj ] ] ]
```

*Ignore j in forward-back-solve, where {A} and {C} are vectors.

Algorithm 2-6: Block Operator $\{A\} = \{A\} - \{B\}^T\{C\}$

```
1.  FOR j = 1 TO L DO*
2.    [ FOR i = 1 TO L DO
3.        [ FOR k = 1 TO i-1 DO  [ A_ij = A_ij - D_ki A_kj ] ;
4.          A_ij = A_ij/D_ii ] ]
```

*Ignore j in forward-solve, where {A} is a vector.

Algorithm 2-7: Block Operator $\{A\} = \{A\}/\{D\}$,
where {D} is a Symmetric Diagonal Block

```
1.  FOR j = L TO 1 STEP -1 DO
2.    [ x_j = x_j/D_jj ;
3.      FOR k = 1 TO j-1 DO  [ x_k = x_k - D_kj x_j ] ]
```

Algorithm 2-8: Back-solve Block Operator $\{x\} = \{x\}\backslash\{D\}$,
where {D} is a Symmetric Diagonal Block

To convert from a scalar to a block algorithm, we replace each scalar operation (subtract-multiply, divide, and square root) by the corresponding block operator specified by Table 2-1 and Algorithms 2-6, 2-7, and 2-8. Operations which always occur together in these algorithms are combined in the block operators. Algorithm 2-9 is the result of generalizing the Cholesky factorization to block form, and Algorithm 2-10 generalizes the forward-back-solve.

Blocks near the edge of the band contain elements which are not within the band, and are therefore padded with zeroes. The block operators can and should be implemented to avoid operating on these extra elements, but there are still I/O costs associated with them. We define the **band pad** to be the difference between the maximum bandwidth that fits a given block structure and the actual bandwidth, which Figure 2-5 shows to be $L\bar{M}-M$. For a given bandwidth, L should be chosen to minimize this band pad and the extra costs associated with storing and transferring these zeroes.

We show operation counts for these operators under the various special cases in Table 2-2. A useful property of block algorithms in the secondary storage context is that the ratio of computation to I/O is not fixed for the block operators. For block size L, the computational cost of each operator is $O(L^3)$, while the time required to transfer blocks is $O(L^2)$. This property allows us to control the relative costs of computation and I/O through the choice of block size.

1. FOR $j = 1$ TO $\bar{N}$ DO
2. [FOR $i = MAX(1, j-\bar{M})$ TO $j-1$ DO
3. [FOR $k = MAX(1, j-\bar{M})$ TO $i-1$ DO
4. [$(A)_{ij} = (A)_{ij} - (U)_{ki}{}^T(U)_{kj}$] ;
5. $(U)_{ij} = (A)_{ij}/(U)_{ii}$;
6. [FOR $k = MAX(1, j-\bar{M})$ TO $j-1$ DO
7. [$(A)_{jj} = (A)_{jj} - (U)_{kj}{}^T(U)_{kj}$] ;
8. $(U)_{jj} = (A)_{jj}^{1/2}$]

Algorithm 2-9: Block Inner-Product Cholesky Algorithm

---

1. FOR $j = 1$ TO $\bar{N}$ DO
2. [FOR $i = MAX(1, j-\bar{M})$ TO $j-1$ DO
3. [$(x)_j = (x)_j - (U)_{ij}{}^T(x)_i$] ;
4. $(x)_j = (x)_j/(U)_{jj}$] ;
5. FOR $j = \bar{N}$ TO 1 STEP -1 DO
6. [$(x)_j = (x)_j\backslash(U)_{jj}$]
7. FOR $i = MAX(1, j-\bar{M})$ TO $j-1$ DO
8. [$(x)_i = (x)_i - (U)_{ij}{}^T(x)_j$]]

Algorithm 2-10: Block Band Forward-Back-Solve, U^TU Factorization

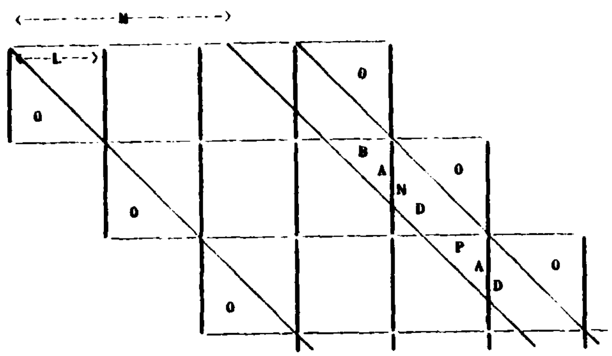


Figure 2-5: The Band Pad of a Block Partitioning

Operator	Approximate number of multiplies		
		C full	C lower-triangular
$\{A\} = \{A\} - \{B\}^T\{C\}$	A nonsymmetric: or A symmetric: ($\{B\}=\{C\}$)	L^3 / $L^3/2$	$L^3/2$ / $L^3/6$
$\{C\} = \{C\}/\{D\}$	D is symmetric:	$L^3/2$	$L^3/6$
$\{D\} = \{D\}^{1/2}$	D is symmetric:	$L^3/6$	

Table 2-2: Multiplication Counts of Block Factorization Operators

CHAPTER 3

The Use and Performance of Paging Systems

3.1 Characteristics of Paging Systems

Virtual memory paging systems offer one solution for solving a problem that does not fit into primary memory. The main advantage of paging is the automatic use of primary and secondary storage, insulating a user from the details of I/O and memory management. However, it is necessary to pay some attention to the characteristics of paging in carrying out large numerical computations in order to avoid unnecessary and sometimes catastrophic increases in execution time. Furthermore, the size of a system that can be solved is constrained by the size of a machine's virtual memory address space. Although this is often very large, the DEC-System 2060 is an *example* of a machine whose virtual memory is smaller than its physical memory.

McKellar and Coffman [23] and Moler [24] made some of the earliest observations concerning the efficient solution of linear systems with paging. Rogers [32] and Trevidi [37] consider this problem in more detail, extending the results to include other numerical algorithms and

techniques such as prepaging. Abu-Sufah [1] has worked on the compiling of programs into code that has good paging characteristics. In this chapter, we present some of the principal techniques for reducing the amount of paging with the factorization algorithms from Chapter 2. We analyze several of these approaches as adapted to banded systems so that, in later chapters, we can compare paging costs with the I/O costs of secondary storage methods.

We attempt to be consistent with terminology from the literature in describing the characteristics and performance of paging systems. In particular, we are interested in the amount of paging that is induced by a program's memory references to the pages containing its data arrays. These data arrays are partitioned sequentially, or <u>paginated</u>, into <u>pages</u> of a fixed size by the virtual memory system. The <u>working set</u> consists of those pages that are <u>active</u>, or in *primary* memory, at a given time. We assume that the <u>working set size</u> (the maximum number of pages in primary memory) is fixed during a computation, and that all pages are initially in secondary storage.

When a page is referenced but is not in the working set, a <u>page fault</u> occurs, in which that page is brought into primary memory from secondary storage. *If* the working set is full, then an active page must be replaced and, if it has been modified while in the working set, rewritten to secondary storage. We shall assume that the page to be replaced is chosen by the <u>least-recently-used</u> (LRU) *criteron* [32].

Since paging usually accompanies time-sharing, a system may choose the LRU page among all programs, but we limit our attention to the case of a *single program with a constant working set size*. We shall *consider* the <u>paging cost</u> of an algorithm to be a count of the page faults incurred plus the number of necessary page writes. We define <u>minimal paging</u> to *mean that each page is read and written only once during a given* algorithm.

Finally, we use the term <u>fragmentation</u> loosely to describe the *costs incurred when elements of a page are transferred and stored*, but not used. Fragmentation can occur for several reasons. For example, a set of data may be padded with zeroes in order to fit a page. Or, an ill-ordered algorithm may reference only a few elements in a page before it is replaced. We shall only discuss the fragmentation inherent in the block storage scheme of Figure 2-4, where the diagonal and band-edge blocks are padded with zeros. The other sources of fragmentation are avoided by assumptions made to simplify the analysis, or because our algorithms are well-ordered.

In order to determine the qualitative characteristics of paging costs, we performed timing experiments with the DEC-System 2060 TOPS-20 *paging system. These experiments consisted of system calls that* manually mapped from 1 to 6 pages at a time in both sequential and random order between disk storage and primary memory. These manual paging costs do not include all of the bookkeeping costs of a page

Pages per command	Sequential order 60 pages CPU	Wall	Per page CPU	Wall	Random order 60 pages CPU	Wall	Per page CPU	Wall
1	190	1020	3.2	17	184	1212	3.1	20
2	167	1137	2.8	19	162	1283	2.7	21
3	150	1068	2.5	18	142	1256	2.4	21
4	137	1055	2.3	18	129	1169	2.2	19
5	123	993	2.1	17	118	989	2.0	16
6	122	985	2.0	16	118	1123	2.0	19

All times in msec., page size = 512 words.

Table 3-1: Timings of DEC-System 2060 Page Map Commands

fault, such as determining the page to be replaced and updating the LRU order. The CPU and wall times for each of these operations are shown in Table 3-1.

The results show that the wall-clock time per page is independent of the number of pages mapped per command. The wall time per page is only marginally higher for random as opposed to sequential order, and for single as opposed to multiple page map commands. The only significant trend is that CPU costs slightly decline as the number of pages per command grows. Thus, the transfer rate of paging on this system does not seem to increase even as we increase the number of consecutive pages being transferred.

The performance of FORTRAN I/O is qualitatively different in that larger records are generally transferred at a higher rate per word. We shall see this in time trials to be reported in Chapter 7.

Consequently, the paging costs of a factorization algorithm qualitatively differ from the I/O costs of a secondary storage method as primary memory usage varies. We shall compare the two approaches in detail in Chapter 5.

In the remainder of this chapter, we discuss paging with factorization algorithms, particularly those presented in Chapter 2 for banded systems. In Section 2, we summarize McKellar and Coffman's landmark results on paging with Gaussian elimination [23]. We then evaluate the performance of LRU paging on forms of the band Cholesky factorization as paginated by columns (Section 3) or by blocks (Section 4). In both cases, there is a working set size threshold above which there is minimal paging, and below which there is paging at a rate that is an order of magnitude higher. Unfortunately, on either side of this threshold, the working set size has almost no effect on the amount of paging. Thus, the availability of more primary memory does not necessarily reduce the amount of paging for these algorithms. These results point to the need for additional tools to be offered by paging systems, or for alternatives to paging, in order to effectively use the available primary memory to minimize I/O costs. In Section 5, we summarize the results and conclusions of this analysis of paging. We also suggest features of a paging system that would allow some control over the length and timing of transfers in order to increase the transfer rate and/or overlap I/O and computation by prepaging.

3.2 Paging with Gaussian Elimination

Much of the paging analysis in the literature is directed at Gaussian elimination for dense nonsymmetric systems, but the principles *apply to other problems as well. For example,* Moler [24] observes that it is important to be aware of possible conflicts between the pagination of matrices and the ordering of operations in a matrix computation. In particular, FORTRAN uses column major storage of matrices, and thus the pagination is by columns. If the inner loop of a computation traverses a *row, then it references every page even though it may perform only* a few operations per page. By reordering an algorithm to traverse columns, we can reduce this paging rate by an order of magnitude.

Locality considerations such as this were incorporated into all of the algorithms and storage schemes presented in Chapter 2. We also identified characteristics of the algorithms that affect paging costs. For instance, the inner-product form of the factorization does not modify subordinate elements, *as the outer-product form does.* Thus, when a page containing only subordinate elements is replaced in the inner-product algorithm, it need not be rewritten to secondary storage.

McKellar and Coffman [23] analyze paging for dense Gaussian elimination with row and submatrix (or block) pagination schemes. The number of "page-pulls" is expressed in terms of P, the number of pages occupied by the coefficient matrix, since the value of P is

approximately the same regardless of the pagination scheme. For row pagination, the number of page transfers in the factorization is $P^2/2$ plus lower-order terms. For submatrix or block pagination, the number of page transfers is $\frac{2}{3}P^{3/2}$ plus lower order terms. In both cases, the working set size affects only lower-order terms. Thus, *small changes in* the working set size have almost no effect on the amount of paging in *dense Gaussian elimination.*

However, their analysis does not consider whether or not a replaced page has been modified and needs to be rewritten. For Gaussian elimination, almost all referenced pages are modified and thus a page pull consists of two page transfers. Also, they do not analyze LRU paging but rather a strategy which determines an optimal page replacement sequence for a given algorithm and working set size.

3.3 Strip Pagination with the Band Cholesky Algorithm

We now examine the paging costs for solving a symmetric, positive definite banded linear system when the matrix is paginated by columns, each page containing a strip. We analyze only the factorization stage, i.e., the band Cholesky algorithm (see Figure 2-5). In a reasonably ordered forward-back-solve, each page is referenced once in each direction so the paging analysis is trivial.

The strip pagination scheme is illustrated in Figure 3-1. We

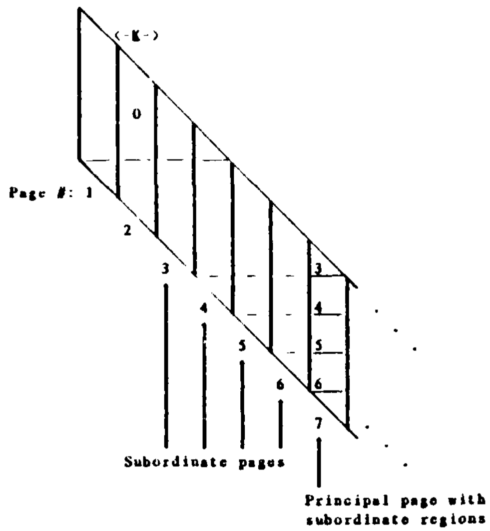


Figure 3-1: Strip Pagination of a Symmetric Band Matrix

---

assume that the page size is $K(M+1)$, so that A is paginated with one strip of K full columns per page. We assume that K is a factor of M, and let $\tilde{M}=M/K$. By Moler's criteria [24], this combination of pagination and computation by columns should result in reasonable paging characteristics. In fact, the paging is as follows.

During the time in which values of U are computed for a given page,

we must reference subordinate elements in the previous $\tilde{M}$ pages. In Figure 3-1, page 7 is divided into regions that are labeled to show which page contains the region's subordinate set. We call these the subordinate regions of the principal page.

If the size of the working set is at least $\tilde{M}+1$, that is, primary memory exceeds $(K+M)(M+1)$, then there is minimal paging with the LRU strategy. We illustrate this for the example of Figure 3-1 with a working set size of $\tilde{M}+1$ or 5.

When page 7 is first referenced, the working set in LRU order is {2,3,4,5,6}. The ensuing page fault replaces page 2, writing it out, and the working set is now {3,4,5,6,7}. As each column of page 7 is computed, the referencing of subordinate pages induces no page faults. The LRU order of the subordinate pages rotates as the computation passes down each column through its subordinate regions, but at the end of each column the working set returns to its original ascending order. Thus each page will be read and written only once during the factorization. A larger working set size is wasteful in that paging is not substantially reduced. Its only beneficial effect is that more pages are active when the factorization is complete, so there will be fewer page faults if a back solve follows.

However, consider what happens when the working set size is reduced by one, to $\tilde{M}$. The first operation with page 7 finds the working set at {3,4,5,6}. Page 7 therefore becomes active, replacing page 3.

Thereafter, each column in page 7 incurs the following page faults, marked with asterisks.

	Computation sequence:	Working Set (LRU order):
	end of 6	{3,4,5,6}
	begin 7	*{4,5,6,7}
For K columns:	7 with 3	*{5,6,3,7}
	7 with 4	*{6,3,4,7}
	7 with 5	*{3,4,5,7}
	7 with 6	*{4,5,6,7}
	finish 7	{4,5,6,7}

The total is $K\tilde{M}+1$ or $M+1$ page faults for each principal page. Also, a page write is required when page 6 is first replaced, since it is modified immediately preceding this sequence. If page 3 is referenced before page 7, then only the first page fault in the loop is avoided the first time through.

This example demonstrates how an algorithm that seems page-local can have a catastrophic paging rate even when the working set size is within one page of good performance. Furthermore, all working set sizes from $\tilde{M}$ down to 2 induce the same amount of paging.

It is possible to reduce this paging rate by a factor of K by modifying the order of operations to observe subordinate page locality. Rather than computing each column as a whole, the algorithm should compute all values of U in one subordinate region of the principal page before proceeding to the next region. We shall call this the <u>strip-local</u> order of operations. Although pagination is by strips, this

ordering resembles block factorization in that the subordinate regions are K by K blocks within the strip. This reduces the number of page faults per principal page to $\tilde{M}+1$, with one page write when page 6 is first replaced. In this case, too, the paging rate remains the same for working set sizes from $\tilde{M}$ down to 2. The ordering strategy requires that K, the number of columns on each page, be known to the algorithm, and that columns do not cross over page boundaries. The implementation of this pagination and ordering scheme is nontrivial, and we know of no codes that observe this ordering with strip pagination.

To summarize, there is minimal paging with strip pagination if there are at least $\tilde{M}+1$ active pages, or $(K+M)(M+1)$ words of memory. Paging is at a higher level (catastrophic without strip-local ordering) if there are $\tilde{M}$ active pages or less. Therefore, when the working set size is other than $\tilde{M}+1$ or 2, extra active pages are being held in primary memory without any reduction in the amount of paging. This means that there are not many combinations of problem size and working set size for which paging is truly efficient in its use of primary memory.

3.4 Block Pagination and Factorization

We next examine paging costs of the band Cholesky algorithm when A is paginated and operated upon by blocks as in Section 2-5. Suppose that each page contains one block of L^2 elements of A and that L is a

factor of M (i.e., the block bandwidth $\tilde{N} = \lceil M/L \rceil = M/L$). The block operations in the block factorization algorithm can be performed in various orders, just as the operations of the scalar algorithm can be ordered by rows or columns, inner- or outer-products, etc. We consider three such orderings:

1. Block-column order refers to that of Figure 2-9, the standard inner-product ordering by columns.

2. Reverse block-column order means that the indexing order of the innermost loops of Figure 2-9 are reversed. The motivation for this ordering is that a principal block immediately becomes a subordinate block in the next block operator, and thus may save a page fault. We incorporate this ordering into a block-based secondary storage method in Chapter 4.

3. Block-row order means that the subordinate blocks from previous block-columns are referenced in row order. The motivation for this ordering is that the top row of subordinate blocks, which are not needed again in the factorization, are referenced first and thus will be the first to be replaced. We specify this order in Figure 3-2.

```
FOR j = 1 TO Ñ DO
{ FOR i = MAX(1,j-Ñ) TO j-1 DO
   { (U)ij = (A)ij / (U)ii ;
     FOR k = i+1 TO j-1 DO
       { (A)kj = (A)ij - (U)ik^T (U)ij } } ;
   (U)jj = (A)jj^(1/2) }
```

Figure 3-2: Block-Row U^TU Symmetric Band Factorization

We determine the paging costs by means of an LRU paging simulation. The simulation keeps track of the working set (the LRU order, which active pages have been modified, etc.), and then, from the sequence of page references generated by a given algorithm, counts the page faults and page writes incurred. We then plot the paging costs as a function of working set size, to be compared with the I/O costs of secondary storage methods in later chapters.

In Table 3-2, we show the simulated paging counts for these three block orderings during the factorization of one block-column where N=6. We plot these results in Figure 3-3, which shows that the three orderings have similar paging characteristics. However, block-column order is best overall in that it has the smallest paging costs as summed over the entire interval. When the working set size is very small, reverse block-column order gains an advantage because of the few page faults it avoids. Block-row order performs best only when the working set size is just below the level required for minimum paging. It performs poorly with a small working set size because the page being modified is not referenced by successive operators, resulting in additional page writes.

Note that the qualitative characteristics of paging performance are the same for block pagination as for of strip pagination. That is, there is minimal paging above a certain working set size, and an order of magnitude more paging below this threshold. However, at each of

Size of Working Set	Page faults plus page writes per block-column		
	Block-Column	Reverse Block-Column	Block-Row
28	14	14	14
27	14	24	29
26	29	34	29
25	35	35	30
24	35	35	31
23	35	35	32
22	35	35	33
21	35	35	34
20	35	35	35
.	.	.	.
.	.	.	.
.	.	.	.
12	35	35	35
11	35	40	45
10	41	41	47
9	43	45	55
8	47	46	57
7	49	49	63
6	52	49	63
5	53	50	65
4	54	50	71
3	55	50	73

Table 3-2: Paging Costs of Block Factorization Orderings, $\bar{N}$=6

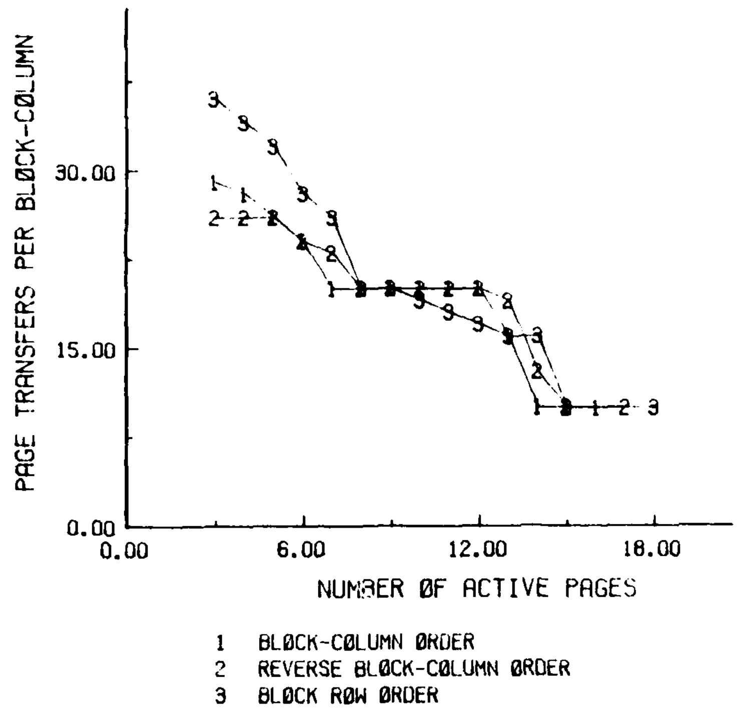


Figure 3-3: Paging Costs of Block Factorization Orderings, $\bar{N}$=6

these levels, the working set size has almost no effect on the amount of paging.

Quantitatively, the threshold for block-column order is a working set size of $(\bar{M}^2+3\bar{M})/2$, i.e., one page less than a principal block-column plus its triangle of subordinate blocks. This working set size occupies

$$M^2/2 + 3ML/2$$

words of primary memory. While this is less than the minimal paging memory requirement for strip pagination, the block algorithm requires more programming effort and computational overhead. Minimal paging is $2\bar{M}+2$ page transfers per block-column. If the page set size is smaller than this, there are additional page transfers for each of the pages containing the triangle of subordinate blocks. This higher level of paging involves $(\bar{M}^2+5\bar{M}+4)/2$ page transfers per block-column. Unless the working set size is very small, paging remains at this level because the principal block-column stays in *memory until it* has been completely factored.

The page size, which dictates the block size, also has an effect on the paging rate for a given bandwidth. This is illustrated in Figure 3-4. For this example, we chose a bandwidth of 150 and simulated the paging for three page sizes, 1444, 625 and 225, corresponding to block bandwidths of $\bar{M}$ = 4, 6, and 10, *respectively. The figure plots the* number of elements transferred per column against primary memory usage;

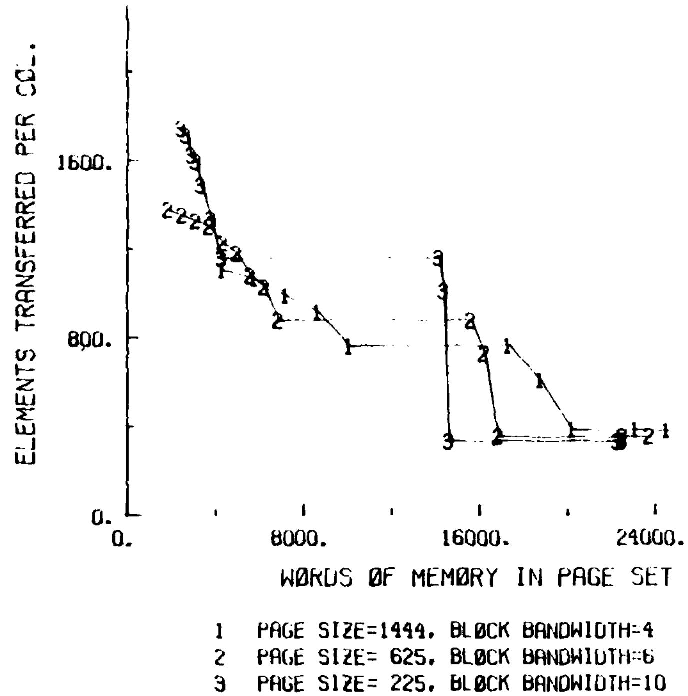


Figure 3-4: The Effect of Page Size on the Paging Rate

thus it is implicit in the comparison that the *time to transfer* a page is proportional to its length. With smaller block sizes, there is less fragmentation in the diagonal and edge blocks, so less primary memory is needed to achieve minimal paging. However, a smaller block size has more paging at the higher paging rate of $O(\bar{M}^2)$.

3.5 Summary of Paging Costs

We now summarize the costs of the paging strategies described in the previous two sections. In order *to compare the costs*, we express the amounts of memory and the number of page transfers in terms of the bandwidth M and the page size, which we denote by S.

For strip pagination, we assumed that there were K columns, or about KM words, per page, so that

$$K=S/M \text{ and } \bar{M}=M/K=M^2/S.$$

From Section 3-3, the amount of memory required for minimal paging is

$$M^2+S \text{ words.}$$

Above this level, there is minimal paging of 2 page transfers per strip, or

$$2M/S \text{ page transfers per column.} \tag{3.1}$$

We showed in Section 3-3 that the the amount of paging below this memory threshold is $\bar{M}+2$ page transfers per strip, or

$$(M^2+2M)/S \text{ page transfers per column.}$$

With *strip-local ordering of operations*, this paging rate can be reduced

to $\bar{M}+2$ page transfers per strip, or

$$(M^3+2MS)/(S^2) \text{ page transfers per column.}$$

For block pagination, we assumed that there was one block of L^2 words per page, so

$$L=\sqrt{S} \text{ and } \bar{M}=M\sqrt{S}/S.$$

From Section 3-4, the amount of memory required for minimal paging is $(\bar{M}^2+3\bar{M})/2$ pages, or

$$(M^2+3M\sqrt{S})/2 \text{ words.}$$

Above this level, there is minimal paging of $2\bar{M}+2$ page transfers per block-column, or

$$2M/S + 2\sqrt{S}/S \text{ page transfers per column.}$$

This is slightly more than (3.1) due to the fragmentation of block pagination. The amount of paging below this level is $(\bar{M}^2+5\bar{M}+4)/2$ page transfers per block-column, or

$$(M^2\sqrt{S}+5MS+2S\sqrt{S})/(2S^2) \text{ page transfers per column.}$$

To graphically compare these costs, we plot page transfers per column versus memory usage in Figure 3-5. The comparison is based on a specific example of bandwidth and page size. The page size is chosen not to be a realistic value, such as 512, but to eliminate any avoidable *fragmentation* with the given bandwidth and strategy. For a bandwidth of 150, we examine strip pagination costs with a page size of 604 (4 full columns per page) and block pagination costs for a similar page size of 625 (block size 25, block bandwidth $\bar{M}=6$).

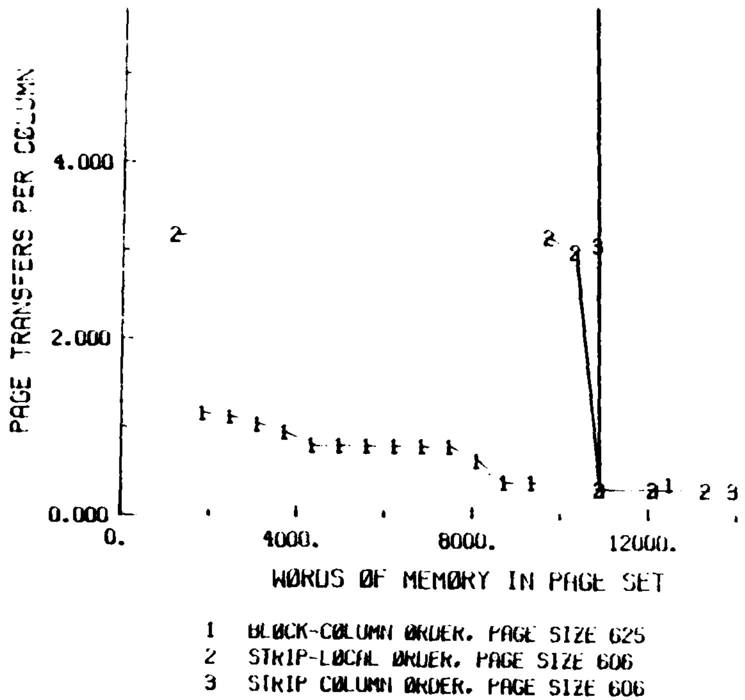


Figure 3-5: Paging Costs vs. Primary Memory Usage

The advantages of paging are its convenience and generality, but we have shown that *it is not necessarily a good means* for solving problems whose storage exceeds the amount of primary *memory*. Among the drawbacks we have mentioned are:

- Virtual memory systems simply do not exist for many machines of interest.
- A machine's virtual memory address space still *limits storage* to what may be an unacceptable level.
- The use of *secondary storage* is conveniently transparent, but beyond the direct control of the user, which may prevent its effective use.
- Under our assumptions, paging does not exhibit a smooth trade-off between memory usage and I/O. Thus the use of more memory does not necessarily improve performance, and paging can go from *optimal to catastrophic* levels with small variations in memory usage.
- The methods that have been shown to considerable reduce high paging levels require considerable programming effort.
- Finally, most general paging systems do not overlap I/O with computation, and techniques such as *prepaging do not* guarantee such overlap, since the sequence of page references cannot be predicted with certainty.

This analysis *of paging performance for band* Cholesky algorithms points out that it would be desirable for a paging system *to recognize* the thresholds of sharp paging increases, and to avoid constraining the working set size below these thresholds whenever possible. Alternatively, the user should be able to specify the optimal size for a *program's* working set in cases where paging can be accurately predicted. Such features would *help to* avoid the dramatic paging increases that occur in these algorithms.

A paging system could be used even more efficiently if it offered features for controlling the length and timing of page transfers. In Chapter 5, we shall show that secondary storage methods use primary memory to reduce I/O costs by transferring as large records as possible in each operation. In Chapter 6, we shall further demonstrate how I/O can be overlapped with computation with secondary storage methods. If a paging system would give the user enough control to use these techniques, a virtual memory system's innate capability for fast I/O between disk and memory could be effectively exploited.

For example, the user could have the option of causing a contiguous set of pages to be transferred at once, which the system should be able to carry out at a higher rate than individual page transfers. In addition, if a transfer could be initiated before the pages are actually needed and the program could continue in the meantime, the overlap of I/O could be achieved. These tools would allow a programmer to use the same techniques for controlling the pagination and transfer of data as would be done with explicit FORTRAN I/O.

That is, we do not wish to say that a paging system cannot use secondary storage efficiently. Rather, the conclusion of this chapter is that passive, inflexible I/O strategies such as LRU paging cannot be as efficient as schemes that control I/O, whether with a paging system or a FORTRAN I/O system. We shall present such schemes in the following chapters.

CHAPTER 4

Secondary Storage Methods

4.1 Introduction

In this chapter, we introduce methods for explicitly using secondary storage with the band algorithms presented in Chapter 2. Each of these methods extends the factorization algorithms by specifying:

- A partitioning of band elements into strips or blocks, which serve as secondary storage transfer records;
- Those records and any other elements which are to be in primary memory at each point in the algorithm, thus determining the primary storage requirement;
- The I/O operations needed to carry out the algorithm within the primary memory constraint by transferring records to and from secondary storage;
- The order of operations that maximizes the strip or block locality of the algorithm.

The methods of this chapter do not address the problem of where records and elements are actually stored within primary memory arrays. As established in Chapter 1, indexing in the algorithms always refers to the position of elements within the full matrix, and it is left as a problem of implementation to transform these indices to the actual

location of the elements in primary memory. We discuss such implementation issues in Chapter 7.

All of our secondary storage methods for the band Cholesky algorithm share certain characteristics. We assume that the elements of b reside in primary memory, and the band elements of A initially reside in secondary storage, partitioned into records as the method requires. Each method reads a record of A, computes the values of U for that record, and writes the records of U to another secondary storage file. The difference between methods is the means by which subordinate elements needed to compute the values of U for a principal record are either stored in primary memory or retrieved from secondary storage.

The forward-solve can be computed using the values of U in the same order as they are computed, so it is carried out simultaneously with the factorization to avoid redundant I/O. We use the term <u>forward pass</u> to refer to this simultaneous factorization and forward-solve. For the back-solve, the records of U must be retrieved in reverse order in a <u>backward pass</u> through the band. This requires an order of magnitude less computation and, as we shall see, less I/O than the forward-pass.

While we have assumed that the right-hand side is in primary memory, we do not include its storage in the analysis of primary memory usage, since it requires the same amount of storage for every method. It could be argued that this is misleading because the storage of the right-hand side would be the dominant storage cost for some of these

methods, and would thus have an effect on some of the theoretical results of later chapters. However, we wish to point out that it is possible to extend the secondary storage methods to include the partitioning and transfer of the right-hand side with only low order amounts of additional I/O. We have implemented such methods, which involve only slightly extra effort. For the sake of simplicity, we shall ignore the costs associated with the right-hand side, which are all of low order.

The strategies we shall consider arise from two types of partitioning, introduced as pagination schemes in Chapter 3. In the first, each record contains a <u>strip</u> of K complete rows or columns of the band of A, as illustrated in Figure 4-1. The second is <u>block</u> partitioning, introduced with block factorization in Figure 2-4, where each record contains an L by L submatrix. In both cases, records are contiguous in primary memory so they can be transferred easily and efficiently.

On the surface, there is no difference between partitioning versus pagination. However, there is a big difference in the constraints posed by their secondary storage environments. In the case of paging, the page size is fixed by the operating system, and thus strip and block size is fixed for a given bandwidth. For secondary storage methods implemented in FORTRAN, there is no practical constraint on the size of I/O records, so we may choose a strip or block size to achieve various

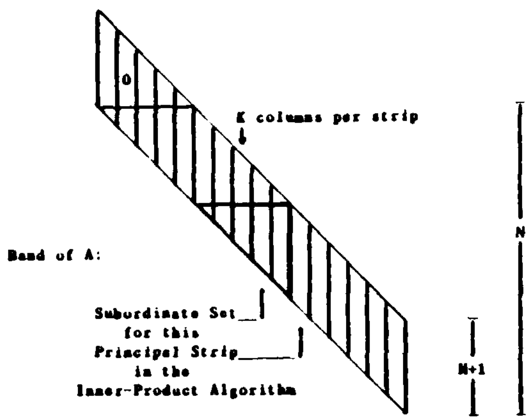


Figure 4-1: Strip Partitioning of the Band of A by Columns

goals. In Chapter 5, we shall show how this flexibility results in a much smoother tradeoff between primary memory usage and I/O costs than that observed for LRU paging in Chapter 3.

In Section 2, we introduce two <u>minimal-I/O</u> secondary storage methods that use strip partitioning. By this we mean that each strip is transferred in and out of primary memory just once during the factorization. The methods require about M^2 and $M^2/2$ words of primary memory to compute the factorization, but the storage scheme of the latter method incurs greater computational overhead.

In Section 3, we define a strip method that reduces the primary

memory requirement to about 2KM words with an order of magnitude more I/O. It uses the strip-local ordering scheme introduced with paging in Chapter 3, and keeps only two arbitrarily thin strips in primary memory at a time.

In Section 4, we present two secondary storage methods based on block factorization. One of them reduces the primary memory requirement to $3L^2$ words, which permits factorization for any bandwidth in a given memory space.

In Section 5, we discuss how the backward pass can be carried out for each partitioning. Throughout the chapter, we shall cite similar secondary storage methods reported in the literature.

4.2 Strip Factorization with Minimal I/O

In the inner-product band Cholesky algorithm, the *elements of a given column* are referenced during at most M+1 successive steps of the factorization algorithm. During the first such step it is the principal column, when the values of U for *that column are computed*. These values then become subordinate elements for up to M succeeding steps. The most obvious reduction in the use of primary memory is accomplished by storing columns in *secondary storage* except during those M+1 steps. This requires I/O operations in only the outermost loop of the algorithm, which results in a minimal-I/O method.

```
 1. FOR j = 1 TO N DO
Comment: This column replaces column j-M-1 of U if j>M+1.
 2.   { INPUT column j of A ;
 3.     FOR i = MAX(1,j-M) TO j-1 DO
 4.       { FOR k = MAX(1,j-M) TO i-1 DO
 5.            [ A_ij = A_ij - U_ki U_kj ] ;
 6.          U_ij = A_ij / U_ii } ;
 7.     FOR k = MAX(1,j-M) TO j-1 DO
 8.       [ A_jj = A_jj - U_kj^2 ] ;
 9.     U_jj = (A_jj)^(1/2) ;
Comment: Retain values in memory until replaced at Line 2.
10.     OUTPUT column j of U }
```

Algorithm 4-1: The Strip-Rectangle (SR) Method, 1 Column per Strip

---

We first present the <u>Strip-Rectangle</u> (SR) method, since subordinate elements are retained in memory in a straightforward rectangular storage *scheme*. Algorithm 4-1 is the SR method in its simplest form, where a strip contains one column. The j^{th} column of A is brought into primary memory at the beginning of the j^{th} step of the factorization algorithm. During this j^{th} step, the values of U for this column are computed and then output, but they are kept in primary memory as they are *referenced* during the next M steps. After the last such reference, the column is free to be overwritten in the next input operation.

This method was described for implementation on vector computers such as the Cray-1 in [20], where they also suggest that in practice, we may not want to perform I/O at every step of the factorization. Algorithm 4-2 is a more general form of the SR method which transfers K columns in and out of primary memory once every K steps. The choice of K allows us to use more memory to reduce the number of I/O events. This method has also been implemented for the CDC-7600 in [15], where provision was made to use that machine's capability to overlap FORTRAN I/O with computation. We shall examine techniques for overlapping I/O with these methods in Chapter 6.

The SR method stores in primary memory the principal strip plus those *preceding* strips that contain subordinate elements. The number of such subordinate strips is $\tilde{M}=\lceil M/K \rceil$. We shall assume that K is chosen to be a factor of M, so that $\tilde{M}=M/K$ and there are exactly M columns within the subordinate strips. Thus, the total amount of primary memory needed to store A and U for the SR method is $(K+M)(M+1)$.

In Figure 4-2, we illustrate the I/O cycle of the SR method. As soon as the values of a *principal strip have all been computed*, this strip is written out to secondary storage at Line 10. The next principal strip is then input into primary memory at Line 2, replacing the K columns in primary memory that no longer contain subordinate elements. The forward pass requires the input and output of each strip exactly *once*, so the number of strip transfers is $2\tilde{N}$ where $\tilde{N}=\lceil N/K \rceil$.

```
1.  FOR j = 1 TO N DO
Comment: The k^th column replaces the k-M-K^th column in memory.
2.      { IF j MOD K = 1 THEN
                   INPUT ((A_ik, i = MAX(1,k-M) TO k),
                          k = j TO MIN(j+K-1,N) ) ;
3.      FOR i = MAX(1,j-M) TO j-1 DO
4.        { FOR k = MAX(1,j-M) TO i-1 DO
5.            [ A_ij = A_ij - U_ki U_kj ]  ;
6.          U_ij = A_ij/U_ii ]  ;
7.      FOR k = MAX(1,j-M) TO j-1 DO
8.        [ A_jj = A_jj - U_kj^2 ]  ;
9.      U_jj = (A_jj)^1/2  ;
Comment: Retain values in memory after output until replaced at Line 2.
10.     IF j MOD K = 0 OR j=N THEN
                 OUTPUT ((A_ik, i=k-M TO k), k=j-K+1 TO j ) ]
```

Algorithm 4-2: The Strip-Rectangle (SR) Method, K Columns per Strip

---

By storing and operating upon elements within their strips, the SR method makes storage, I/O, and the computational algorithm as simple as possible. However, it does so at the cost of storing elements in primary memory that are actually no longer necessary in the forward pass. The figu.es illustrate that only half of the elements of U being retained in primary memory are actually subordinate elements. Thus, we could achieve a minimal-I/O method within less primary memory by using a triangular storage scheme to retain the subordinate elements.

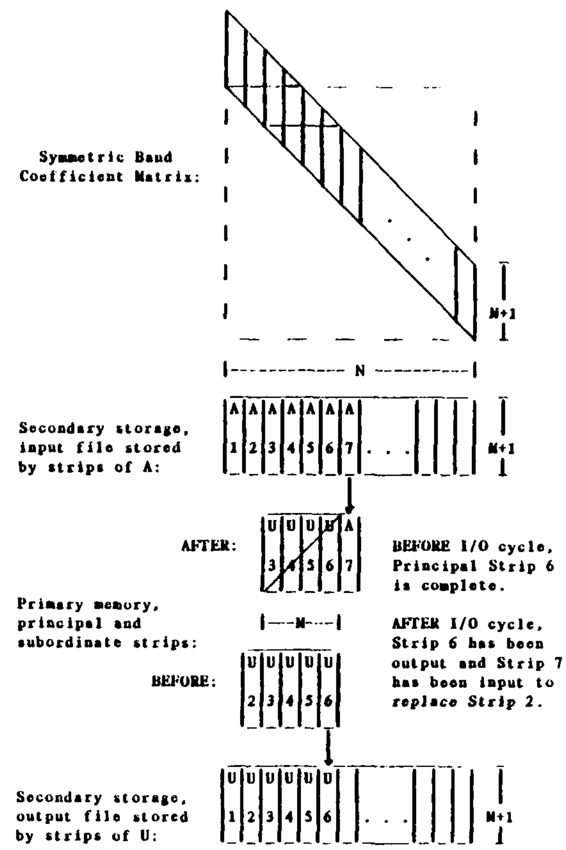


Figure 4-2: I/O-Storage Scheme for SR Method

Variations of this approach, which we call the <u>Strip-Triangle</u> (ST) method, have been described in the literature [29, 34]. In Figure 4-3, we illustrate the I/O and storage schemes of the ST method for the inner-product Cholesky algorithm. After reading in a strip of K columns, we compute U for one principal column at a time, simultaneously shifting subordinate elements within the triangle to prepare for the next column, as shown in Figure 4-4. *The principal elements themselves are shifted into the triangle since they are subordinate elements for the next column.* Since the subordinate elements needed for subsequent strips have been shifted into the triangle, the principal strip may be output and immediately overwritten by the next strip.

The triangular scheme for storing the subordinate set reduces the primary memory requirement to (K+M/2)(M+1), with the same amount of I/O as the rectangular scheme. Alternatively, we could use the same amount of primary memory as the SR method to reduce the number of strips (and strip transfers), since K can be chosen with ST to be M/2 larger. However, this method has more overhead than the SR method due to the shifting and rewriting of subordinate elements. *We show the extent of this overhead in the timing experiments of Chapter 7.*

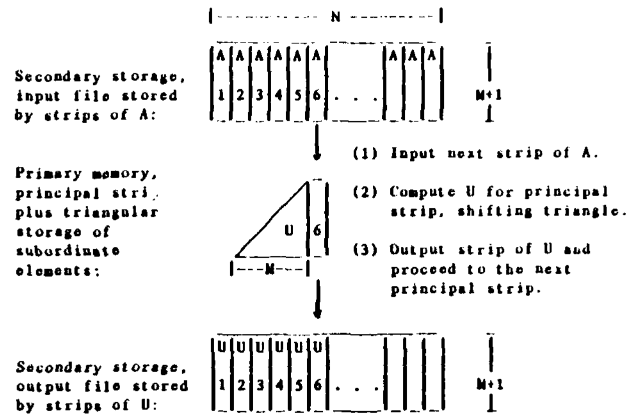


Figure 4-3: I/O-Storage Scheme for ST Method

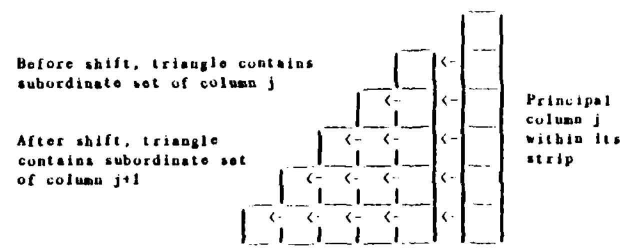
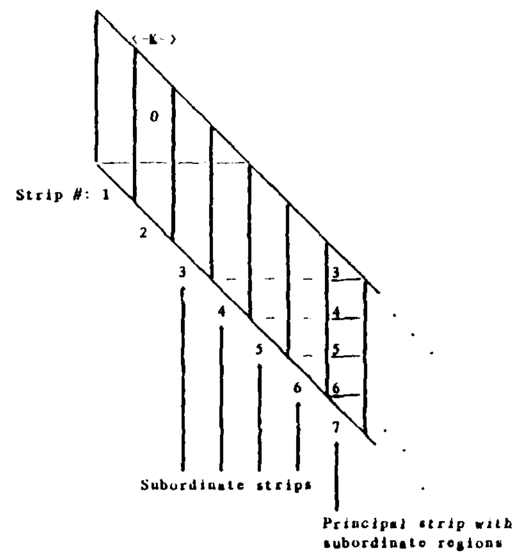


Figure 4-4: Shift of Subordinate Elements in ST Method

4.3 A Strip Method with Subordinate I/O

We next consider several alternatives for reducing the primary memory requirement even further. This implies that we cannot keep all subordinate elements of a principal column in primary memory, and some additional I/O will be necessary to *retrieve* them from secondary storage. In this section we introduce such a method based on strip partitioning. It retrieves the subordinate strips one at a time from secondary storage as the principal strip is computed. The additional I/O can be limited to about $\tilde{N}\tilde{M}$ extra strip transfers during the forward pass if operations are reordered to reflect subordinate strip locality.

Figure 4-5 shows the example from Chapter 3 of a band section partitioned into strips in which $\tilde{M}$=4. Recall that the regions of the principal strip are numbered to indicate the subordinate strip associated with that region. That is, to compute the factorization of region P3 involves elements from strip S3, as well as from regions P1 and P2. If we compute U for the principal strip by columns, each column *requires* the input of all subordinate strips. As with paging, it would be much better to factor the principal strip by *subordinate regions* instead. Thus U is computed for the entire principal strip with $\tilde{M}$ additional strip transfers.



For each Principal Strip:
- (1) Input values of A in Principal Strip (P1 to P5)
- (2) For each Subordinate Strip:
  - (Input S1, Compute values of U in P1)
  - (Input S2, Compute values of U in P2)
  - (Input S3, Compute values of U in P3)
  - (Input S4, Compute values of U in P4)
- (3) Compute values of U in P5
- (4) Output values of U in Principal Strip

Figure 4-5: Computation and I/O for Strip-Strip Method

```
Comment: "A(I)" and "U(I)" refer to Strip I, containing columns
         (I-1)K+1 to IK.  Upper-case indices refer to strips, and
         lower-case indices refer to individual elements.
FOR J = 1 TO Ñ DO
[ INPUT A(J) ;
  jmin = (J-1)*K+1 ; jmax = MIN(J*K,N) ;
  FOR I = MAX(1,J-M̃) TO J-1 DO
Comment: Input Subordinate Strip I and compute U for those
         elements of Strip J with subordinate elements in Strip I.
    [ INPUT U(I) ;
      FOR j = jmin TO MIN(jmax,I*K+M) DO
       [ FOR i = MAX(1,j-M,(I-1)*K+1) TO I*K DO
          [ FOR k = MAX(1,j-M) TO i-1 DO
             [ A_ij = A_ij - U_ki U_kj ] ;
            U_ij = A_ij/U_ii ] ] ] ;
Comment: Compute final region and output Principal Strip J.
    FOR j = jmin TO jmax DO
     [ FOR i = jmin TO j-1 DO
        [ FOR k = MAX(1,j-M) TO i-1 DO
           [ A_ij = A_ij - U_ki U_kj ] ;
          U_ij = A_ij/U_ii ] ;
       FOR k = MAX(1,j-M) TO i-1 DO
          [ A_jj = A_jj - U_kj^2 ] ;
       U_jj = A_jj^(1/2) ] ;
    OUTPUT U(J) ]
```

Algorithm 4-3: The Strip-Strip (SS) Method

We give the details of this method in Algorithm 4-3. Similar methods have been described in the literature [25, 33, 39], but most of them use an outer-product algorithm which modifies subordinate strips and thus requires that they be rewritten to secondary storage.

The amount of primary memory occupied by the two strips is $2K(M+1)$. The I/O required for a full strip is the input and output of the principal strip and the input of $\tilde{M}$ subordinate strips, a total of $\tilde{M}+2$ strip transfers.

4.4 Block Factorization with Secondary Storage

In Chapter 2 we presented algorithms for performing block factorization, forward- and back-solve. Block partitioning leads quite naturally to secondary storage strategies. Since we have defined the block algorithms in terms of localized block operators, a block serves as a unit *for computation* and *for I/O*. In this section we describe two such strategies. We should mention that these block methods are impractical without the capability for random access I/O. The initial input of blocks of A and the output of U are in sequential order, but the retrieval of subordinate blocks of U would be impractical without random access.

We first define a Block-Minimum (or BM) storage and I/O strategy. In block factorization (Algorithm 2-9), the greatest number of blocks

used by a single block operator is three, for the multiply-subtract operator. If we store in primary memory only those blocks involved with the current operator, we have a method that uses at most $3L^2$ *words of primary memory for* block size L. In Algorithm 4-4, we specify the I/O operations needed to carry out block factorization in this manner.

The <u>RELEASE</u> *statement* is used to indicate that a block is not currently needed in primary memory and can be replaced with another block. This occurs after a block has been used by one operator and will not be used by the next operator. *As noted in the comments*, a principal block $(U)_{ij}$ is not immediately released after output. Because of the reverse order of the *loop indices (Lines* 4 and 13), this principal block will be a subordinate block during the next operator, so it is kept in primary memory to avoid rereading it. A close examination of the algorithm will verify that the only blocks in primary memory (i.e., input and not yet released) are those needed by the current block operator.

1. For $j = 1$ to $\bar{N}$ do

2. [For $i = \mathrm{MAX}(1, j-\bar{M})$ to $j-1$ do

Comment: *Compute U for a principal block.*

3. [INPUT $(A)_{ij}$;

4. For $k = i-1$ to $\mathrm{MAX}(1, j-\bar{M})$ step -1 do

5. [INPUT $(U)_{ki}$;

Comment: When $k=i-1$, U_{kj} is already in primary memory.

6. If $k \neq i-1$ then INPUT $(U)_{kj}$;

7. $(A)_{ij} = (A)_{ij} - (U)_{ki}{}^T(U)_{kj}$;

8. RELEASE $(U)_{ki}$, $(U)_{kj}$] ;

9. INPUT $(U)_{ii}$;

10. $(U)_{ij} = (A)_{ij}/(U)_{ii}$;

Comment: Keep $(U)_{ij}$ in primary memory to use at Line 7 or 15.

11. OUTPUT $(U)_{ij}$; RELEASE $(U)_{ii}$] ;

Comment: Similarly, input diagonal block and compute U.

12. INPUT $(A)_{jj}$;

13. For $k = j-1$ to $\mathrm{MAX}(1, j-\bar{M})$ step -1 do

Comment: When $k=j-1$, $(U)_{kj}$ is already in primary memory.

14. [If k=≠j-1 then INPUT $(U)_{kj}$;

15. $(A)_{jj} = (A)_{jj} - (U)_{kj}{}^T(U)_{kj}$;

16. RELEASE $(U)_{kj}$] ;

17. $(U)_{jj} = (A)_{jj}^{1/2}$;

18. OUTPUT and RELEASE $(U)_{jj}$]

Algorithm 4-4: The Block-Minimum (BM) Method

The I/O for the factorization of a full column of $\bar{M}+1$ blocks includes: reading and writing the $\bar{M}+1$ principal blocks at Lines 3, 11, 12, and 18; reading $\bar{M}(\bar{M}+1)/2$ blocks of the second subordinate set at Lines 5 and 9; and reading $(\bar{M}-1)\bar{M}/2$ blocks of the first subordinate set at Lines 6 and 14. Thus the total number of block *transfers required to* compute a full block-column is

$$\bar{M}^2 + 2\bar{M} + 2. \tag{4.1}$$

While this can be a high level of I/O, we have reduced the primary memory required to store A and U to an arbitrarily small amount. Furthermore, as previously mentioned, the ratio of computation to I/O in the block operators is proportional to L and thus the computation will dominate the I/O for L sufficiently large. We shall discuss this in more detail in Chapters 5 and 6. Finally, notice that the BM method could easily be adapted for dense and/or nonsymmetric algorithms within the same amount of primary memory, since all forms of block factorization use at most three blocks per operation.

A second block strategy uses more storage to avoid nearly half of the I/O of the BM method. If we keep all blocks from block-column j in primary memory during the entire j^{th} step of the algorithm, the first subordinate set remains in primary memory and thus the input *operations* at Lines 6 and 14 of Algorithm 4-4 can be eliminated. This is the approach of the <u>Block-Column</u> (BC) method of Algorithm 4-5, which stores a total of $\bar{M}+2$ blocks in primary memory. The factorization of one

1. For j = 1 to N do

Comment 1: Input the jth column of blocks.
2. [INPUT $((A)_{ij}$, i = MAX(1,j-$\bar{M}$) to j) ;
3. For i = MAX(1,j-$\bar{M}$) to j-1 do
4. For k = MAX(1,j-$\bar{M}$) to i-1 do

Comment 2: Input the second subordinate block.
5. [INPUT $(U)_{ki}$;
6. $(A)_{ij} = (A)_{ij} - (U)_{ki}{}^T(U)_{kj}$;
5. RELEASE $(U)_{ki}$] ;
7. INPUT $(U)_{ii}$;
8. $(U)_{ij} = (A)_{ij}/(U)_{ii}$] ;
9. [For k = MAX(1,j-$\bar{M}$) to j-1 do
10. { $(A)_{jj} = (A)_{jj} - (U)_{kj}{}^T(U)_{kj}$ } ;
11. $(U)_{jj} = (A)_{jj}^{1/2}$;
12. OUTPUT and RELEASE ($(U)_{ij}$, i = MAX(1,j-$\bar{M}$) to j)]

***Algorithm 4-5:** The Block-Column (BC) Method*

---

block-column requires the input and output of the $\bar{M}+1$ principal blocks at Lines 2 and 12, and the input of $\bar{M}(\bar{M}+1)/2$ subordinate blocks at Lines 5 and 7. The total number of block transfers for a block-column is therefore

$$\bar{M}^2/2 + 5\bar{M}/2 + 2 \ . \tag{4.2}$$

We should point out that the block methods have a practical upper limit on the block size L and thus on the amount of primary memory used.

For example, if L equals M/2 then 2/3 of the blocks are only half full of elements, being on the diagonal or the edge of the band. To carry out the BM method with this block size requires $3M^2/4$ words of primary memory, which is more than enough for minimal I/O by the ST method. For such a relatively large block size to be useful, we would have to devise a means for storing non-full blocks without padding, which complicates the storage, I/O and computational schemes. *Therefore, the largest* block size we shall use is L=M/3.

4.5 Back-Solving with Strip and Block Methods

The back-solve involves considerably less computation and I/O than the factorization. There is just one multiply per element of U during the back-solve (as opposed to an average of M/2 multiplies per element in the factorization), and it is simple to order the operations of the algorithm by strips or blocks. Thus we can implement the back-solve by reading the strips or blocks into primary memory one at a time in reverse order, and performing all computation with each record before proceeding to the next.

For the SR, ST, and SS methods, where U is stored by columns, the best form for the back solve *performs* outer-products over columns. We show this in Algorithm 4-6.

1. FOR $j = N$ TO 1 STEP -1 DO
2. [IF j MOD $K = 0$ OR $j = N$ THEN
   INPUT ((U_{ik}, $i=MAX(1,k-M)$ TO j), $k=j-K+1$ TO j) ;
3. $x_j = y_j/U_{jj}$;
4. FOR $i = MAX(1,k-M)$ TO j DO
5. [$y_i = y_i - U_{ij}x_j$]]

Algorithm 4-6: The Strip Back-Solve by Columns

---

1. FOR $j = \bar{N}$ TO 1 STEP -1 DO
2. [INPUT $\{U\}_{jj}$;
3. $\{x\}_j = \{x\}_j \Omega \{U\}_{jj}$]
4. FOR $i = MAX(1,j-\bar{M})$ TO $j-1$ DO
5. [INPUT $\{U\}_{ij}$;
6. $\{x\}_i = \{x\}_i - \{U\}_{ij}^T\{x\}_j$]]

Algorithm 4-7: The Block Back-Solve

The back-solve for block methods is similarly straight-forward. In Algorithm 4-7, we show a block back-solve algorithm which needs just one block in primary memory at a time. It is in block outer-product form although the block operators themselves perform inner-products. The loop at Lines 4-6 could be performed in reverse order if it were desirable to input the blocks strictly in reverse order.

In Chapters 5 and 6 we analyze in more detail the relative costs of computation and I/O for the forward and backward passes. *For now we* simply state that amount of I/O in the backward pass, while less than that of the forward pass, is more significant in relation to the amount of computation. Nevertheless, the timings of Chapter 7 will show that the forward pass dominates the total time of solution with secondary storage methods.

4.6 Further Remarks

We assumed in defining secondary storage methods that an input file exists with the coefficient matrix partitioned according to the method's needs. In some cases, it may be possible and desirable to have the strips or blocks of the coefficient matrix generated directly in primary memory by a subroutine. The frontal method [17] uses this approach for performing finite-element simulations using *secondary storage*. *In* the simplest cases, such as the model problem introduced in Chapter 1, the strips or blocks of A are identical to one of a few basic forms. This allows a further simplification by keeping one of each form in primary memory, to be copied into the computational work space when a new strip or block is needed.

We will discuss such user-interface issues further in Chapter 7. For now, we simply point out that some of the costs of these methods might be avoided for certain problems. We shall continue to assume that

the appropriate input file exists and include the initial input of A in the analysis of I/O costs.

The *secondary storage* methods we have introduced for the band form apply with varying success to other types of matrices. For example, if A is dense and nonsymmetric, the factorization of the first row or last column references every element of the matrix. Thus a minimal-I/O method would require the entire matrix in primary memory. However, the SS method and the block methods would yield substantial storage reductions.

All of the methods have analogous approaches that can be applied to the nonpivoting LU factorization of a nonsymmetric banded matrix. The minimal-I/O strip methods are also easily extended to the LU factorization of a *banded* matrix that requires pivoting, since the subordinate elements involved in the pivot search and row or column exchange are in primary memory. However, the block methods are impractical for any algorithm with standard row or column pivoting because a pivot search or exchange would involve many block transfers to perform very little computation [23].

The approach of the SS method has also been applied to profile matrices [25], but the I/O costs depend highly upon the specific profile structure. A profile matrix with a narrow bandwidth, such as that shown for the model problem in Figure 1-1, could certainly be factored with *less computation* and *I/O* than the corresponding band matrix.

Finally, there is a widely expressed need [13] for a general sparse factorization code that uses secondary storage. The complexity and variety of sparse data structures are the main obstacles to fulfilling this need.

CHAPTER 5

Analysis of Costs for Secondary Storage Methods

5.1 Introduction

The secondary storage methods of Chapter 4 require substantially less primary memory than in core methods for factoring banded symmetric, positive definite matrices, but other costs are introduced by the I/O. In this chapter, we characterize these costs with a simple model, and compare the costs associated with the secondary storage methods. We do not recommend a specific method over others, because the criteria for choosing a method can vary. The analysis does illustrate which methods have the least I/O under various conditions, what constitutes a good strip or block size, and how to minimize memory occupancy. In Chapter 6, we shall use the same models to derive the conditions and requirements for overlapping I/O with computation.

In most charging algorithms, there are three main costs associated with the execution of a program that uses secondary storage: CPU, memory occupancy, and I/O. The CPU cost is measured by the CPU clock time devoted to that program. Memory occupancy charges vary between systems,

but the most common measure is the amount of memory occupied multiplied by the time of occupancy. Both computation and I/O contribute to this by increasing the time of occupancy. Measurement of the cost of paging or explicit I/O varies the most between different machines. Among these I/O measures are: time consumed by I/O according to the CPU rate or some other rate; a fixed charge per page fault or I/O event; or no explicit charge at all besides the indirect effect on memory occupancy. Any charges for secondary storage occupancy are insignificant over the duration of a program's execution.

First, we consider CPU costs. Multiplication counts are the simplest way of accurately measuring the amount of computation in numerical algorithms, since the computational time to execute a given program segment tends to be proportional to the number of floating-point multiplies performed. That is, if we define μ to be the CPU time required for the execution of an algorithm divided by the total number of floating-point multiplies then the CPU time required for any n multiplies within the algorithm is $n\mu$.

The value of μ depends not only on the speed of the processor in performing multiplies but also on the overhead of the algorithm. In practice, timings show that there is a comparable amount of overhead in the factorization or forward-back-solve algorithms for nonsymmetric or symmetric, dense or banded matrices. However, there is additional CPU overhead in the block methods due to subroutine calls, indirect array

addressing, etc., and in the Strip-Triangle method due to the shifting and rewriting of subordinate elements. The significance of this overhead depends as much on the design of the machine and compiler as on the method. Since we shall use μ to compare computation and I/O time within each method, and not between methods, we can consider it to be a method-dependent value. In Chapter 7, *we shall show the extent of this* overhead in timing experiments.

We intend for this measure to represent only the CPU costs of computation in an algorithm, *and not those of I/O events. The I/O costs* are obviously not proportional to the number of multiplies. Furthermore, some architectures (array processors for example [2]) can have independent processors to handle memory addressing, sending of disk controller instructions, or other I/O-related tasks. These costs can be represented by the I/O model.

For measuring I/O costs, we are most interested in the wall-clock or turn-around time required to move elements between primary and secondary memory. We assume that the transfer time for a record of n elements is a linear function,

$$T[n] = \sigma + n\tau \, , \qquad (5.1)$$

where σ is the startup time and τ the transfer time per word. These constants can have various meanings, including the hardware parameters

associated with a secondary storage device. For example, with disk transfers, σ would be the seek time for the head to move to the correct track plus the latency time for the disk to rotate to the record.

Our analysis also allows this model to represent other costs, since it is a reasonable assumption that all costs related to I/O depend only upon the number of I/O events and the number of elements transferred. *For example*, we can define σ and τ and experimentally determine their values so that formula (5.1) represents I/O-related CPU time. Or, if a there is a system charge per I/O event regardless of length, then σ is the cost per event and τ=0. Thus, the analysis that follows may have various meanings depending on how σ and τ are defined. Unless otherwise stated, we use the model to represent wall time.

This I/O model assumes that there is freedom to choose any record size, and that the transfer of such a record incurs only one start-up cost, independent of record length. This may not be precisely true in *practice*, since an I/O system usually maps such records into physical disk records which may not be in contiguous locations. However, the model does describe the general characteristics of secondary storage transfers and does seem to characterize the *performance of disk I/O* in practice.

Experimental timings conducted with FORTRAN I/O on a DEC-System 2060 (reported in Chapter 7) show that words within longer records can be transferred at a higher rate per word in spite of the fixed disk record size. This is a fundamental difference between the performance of paging and explicit I/O. With paging, where transfers are of the same length, the system uses more memory to retain more active pages and hopefully avoid page faults. In Chapter 3, we showed that this is not effective with the band Cholesky factorization. With secondary storage methods, we can use more memory to transfer longer records and thus achieve a higher transfer rate. The analysis of this chapter will show how this difference affects the tradeoff between primary memory usage and I/O costs.

In Section 2, we express the amount of I/O in each secondary storage method as a function of the bandwidth and strip or block size. These functions are used to show the relationship between primary memory usage and I/O for a given bandwidth. As a result, we find that there is a much smoother tradeoff between memory and I/O with secondary storage methods than with paging. Furthermore, we can express these costs so that the methods are asymptotically equivalent, differing only in levels of fragmentation and limitations in their ranges of primary memory usage.

In Section 3, we evaluate the amounts of fragmentation in the various methods. Although some fragmentation is avoided by choosing K

or L to be a factor of M, the SS, BC, and BM methods have unavoidable fragmentation.

In Section 4, we derive expressions for the asymptotic levels of memory occupancy for the secondary storage methods, and compare them with those of various in core methods. We find that the memory occupancy of the BM method is as low, asymptotically, as that of any direct or iterative method for solving the linear systems arising from the model problem.

5.2 I/O Functions for Secondary Storage Methods

We now characterize the amounts of I/O in each of the secondary storage methods using the counts of strip or block transfers from Chapter 3, and the formula (5.1) for the cost of each transfer. The result is an <u>I/O function</u>, derived by examining the transfers required for a principal strip or block-column and dividing by K or L. This represents the I/O cost per column, expressed in the form $\sigma C+\tau D$, where C is the number of I/O events and D is the total number of elements transferred. The coefficients C and D depend only on the bandwidth M and the strip size K or block size L. Expressions are simplified by assuming that M is large, and approximating M+1 by M.

The Strip Rectangle and Strip Triangle methods require a given strip to be input and output in the forward pass and input again in the backward pass. Each strip contains K full rows or columns, or about KM elements, so the I/O function for SR or ST is

$$3T[KM]/K = \sigma(3/K) + \tau(3M).$$

The Strip-Strip method had $\tilde{M}+2$ transfers per principal strip in the forward pass and one in the backward pass, so its I/O function is

$$(3+\tilde{M})T[KM]/K = \sigma(\tilde{M}+3)/K + \tau M(\tilde{M}+3).$$

The number of block transfers per block-column in the Block Minimum forward pass was given by equation (4.1) as $\bar{M}^2+2\bar{M}+2$. Adding M+1 block transfers in the backward pass, the I/O function for BM is

$$(\bar{M}^2+3\bar{M}+3)\ T[L^2]/L = \sigma(\bar{M}^2+3\bar{M}+3)/L + \tau L(\bar{M}^2+3\bar{M}+3).$$

For the Block-Column method (using equation (4.2)), the I/O function is

$$(\bar{M}^2/2 + 7\bar{M}/2 + 3)\ T[L^2]\ /\ L = \sigma(\bar{M}^2+7\bar{M}+6)/(2L) + \tau L(\bar{M}^2+7\bar{M}+6)/2.$$

In Table 5-1, we summarize these I/O functions along with the primary memory requirements of the methods. The SR memory requirement assumes that K evenly divides M, but there are no other constraints on K or L. These expressions allow us to compare the methods, but the results of this comparison depend on the relative values of σ and τ. If σ is large, it is better to transfer large records in order to reduce the number of I/O events. If σ is small relative to τ, then the number of elements transferred is the significant factor. This discourages large records that may result in more fragmentation. We examine the two extremes by separately plotting the coefficients of the σ and τ terms versus primary memory usage.

Secondary Storage Method	Primary Memory Used for A	I/O per Column
SR ST	$(K+M)M$ $(K+M/2)M$	$\sigma(3/K) + \tau(3M)$
SS	$2KM$	$\sigma(\bar{M}+3)/K + \tau M(\bar{M}+3)$
BC	$(M+2)L^2$	$\sigma(\bar{M}^2+7\bar{M}+6)/(2L)$ $+ \tau L(\bar{M}^2+7\bar{M}+6)/2$
BM	$3L^2$	$\sigma(\bar{M}^2+3\bar{M}+3)/L$ $+ \tau L(\bar{M}^2+3\bar{M}+3)$

Table 5-1: Primary Memory and I/O Requirements of Secondary Storage Methods

In Figures 5-1 and 5-2, we plot the coefficients of σ and τ for the I/O functions over a range of strip and block sizes, and with a bandwidth of 100. Each I/O function is a weighted sum of these two coefficient plots with σ and τ the weights. The SR method is not included because its I/O costs are the same as for the ST method which requires less primary memory. The advantage of SR is computational efficiency as we shall see in the experimental timings of Chapter 7.

The figures illustrate several points. The first is that several

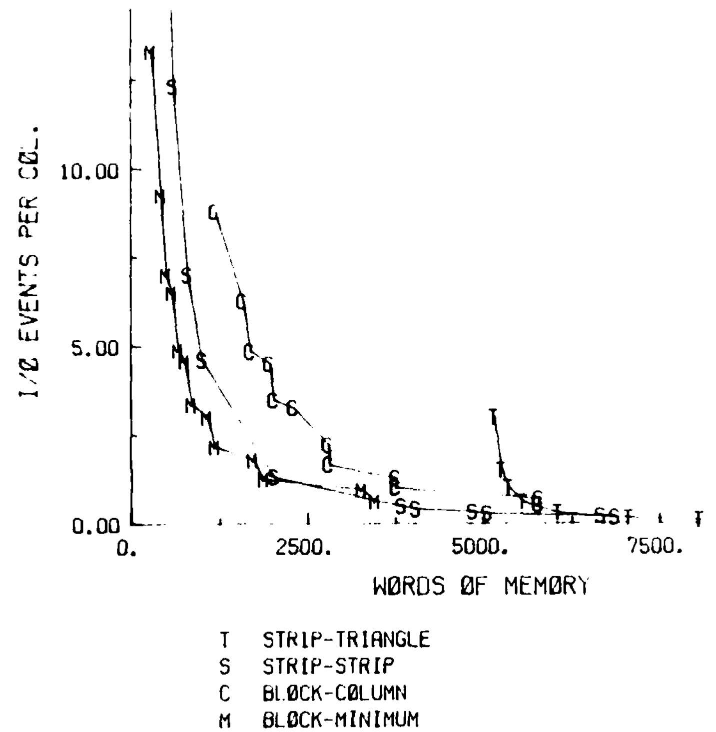


Figure 5-1: σ Coefficient vs. Primary Memory Usage, M-100

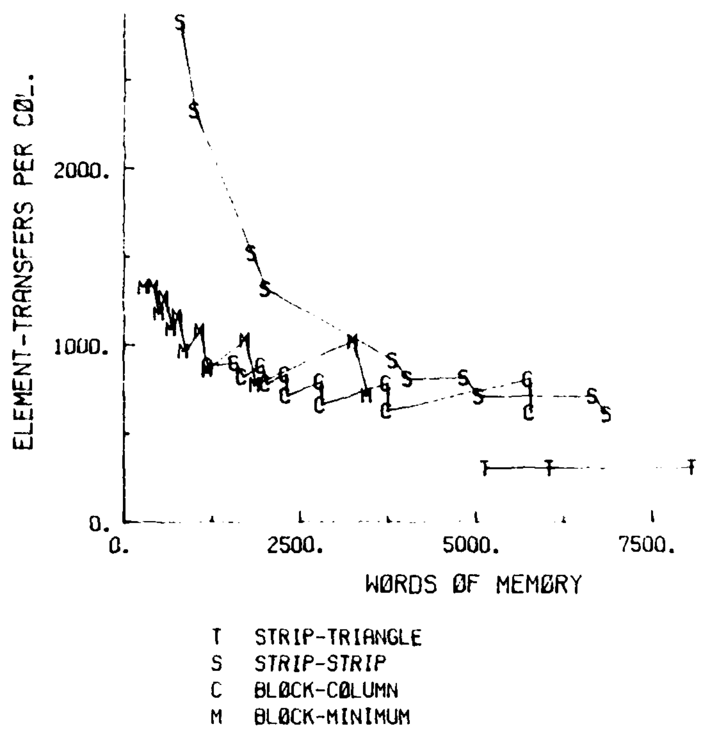


Figure 5-2: τ Coefficient vs. Primary Memory Usage, N=100

of the functions are not smooth. In the block methods, there are dropoffs in the σ coefficient and a sawtooth effect in the τ coefficient that are due to fragmentation. Each "tooth", or upgrade, in the τ function corresponds to a particular block bandwidth M where the band pad (see Figure 2-5) grows with L. The σ and τ dropoffs occur when an increase in block size causes the block bandwidth to decrease, i.e., the band fits within fewer blocks. As a result, a "good" block size is one that minimizes the band pad within a given block bandwidth. That is, for a given $\bar{M}$, L should be the smallest integer *greater than or equal to* $M/\bar{M}$. If $L\bar{M}=M$, then there is no band pad. There is similar, less serious fragmentation in the τ coefficient of the SS method when the trailing subordinate strip contains columns outside the subordinate set.

To draw further conclusions, we examine the same I/O functions using only good choices for K and L, which result in the smooth functions of Figures 5-3 and 5-4. Not unexpectedly, the minimum-I/O *ST* method has the least I/O unless σ is large and K is small. The other methods have similar σ functions, with the SS method practical over the widest range, and the BM method best over its low range of memory usage. The block methods have a clear advantage over SS in the τ coefficient because of less fragmentation, as *the next section will show*.

Suppose that $\bar{N}$ and $\bar{M}$ are held constant as M varies. Thus, K and L are no longer independent variables, and the I/O functions take the form shown in Table 5-2. In this form, the primary memory requirements of

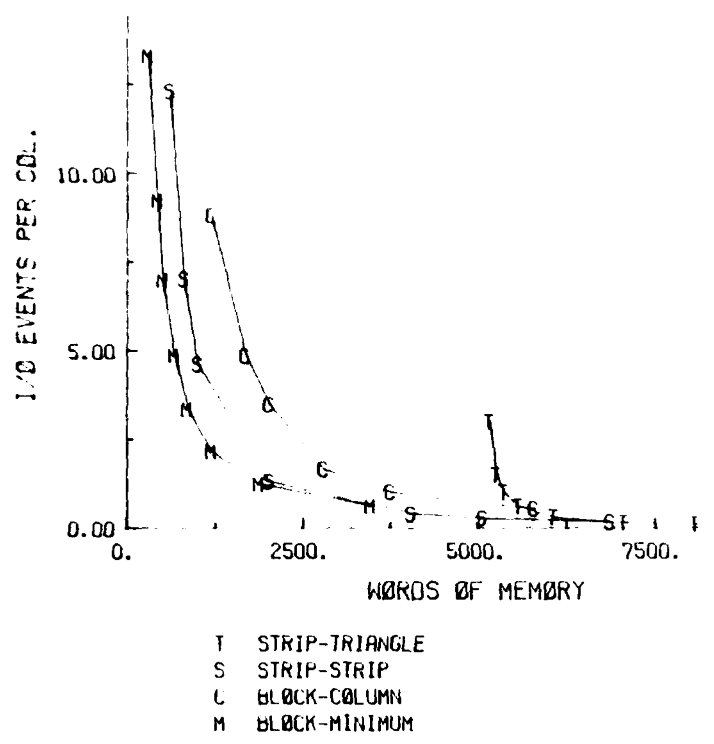


Figure 5-3: σ vs. Memory with "Good" Strip and Block Sizes

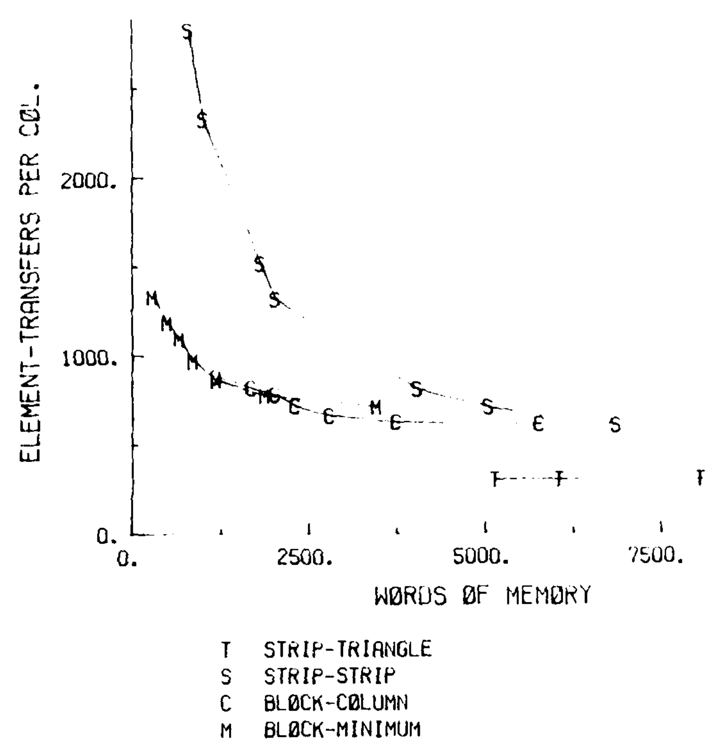


Figure 5-4: τ vs. Memory with "Good" Strip and Block Sizes

all methods are $O(M^2)$, the σ coefficients are $O(M^{-1})$, and the τ coefficients are $O(M)$. This shows that the comparison of I/O costs between methods is not affected by bandwidth. It also shows that all the secondary storage methods have asymptotically equivalent costs, although the coefficients limit the ranges of memory usage. Thus the methods all offer the same asymptotic tradeoff between primary memory usage and I/O costs.

Secondary Storage Method	Primary Memory Used	I/O Cost per Column	
		σ component	τ component
SR	$M^2(1+\tilde{M})/\tilde{M}$	$3\tilde{M}/M$	$3M$
ST	$M^2(2+\tilde{M})/(2\tilde{M})$		
SS	$M^2(2/\tilde{M})$	$(\tilde{M}^2+3\tilde{M})/M$	$M\tilde{M}+3$
BC	$M^2(\tilde{M}+2)/\tilde{M}^2$	$(\tilde{M}^2+7\tilde{M}+6)\tilde{M}/(2M)$	$M(\tilde{M}^2+7\tilde{M}+6)/(2\tilde{M})$
BM	$M^2(3/\tilde{M}^2)$	$(\tilde{M}^2+3\tilde{M}+3)\tilde{M}/M$	$M(\tilde{M}^2+3\tilde{M}+3)/\tilde{M}$

Table 5-2: *Primary Memory and I/O Requirements* with $\tilde{M}$ and $\hat{M}$ Constant

5.3 An Analysis of Fragmentation

We have repeatedly mentioned that various *types of fragmentation* have a major effect on the I/O costs of secondary storage methods. Certain fragmentation costs can be eliminated by choosing K or L to be a factor of M. But there is still unavoidable fragmentation due to unnecessary elements being transferred within subordinate strips in the SS *strategy*, and band padding with the BM and BC methods. In this section, we quantify and compare those unavoidable fragmentation costs. This analysis helps to verify some of the statements made in the previous sections concerning the amount of wasteful I/O in the methods.

Let us define the <u>fragmentation ratio</u> (FR) of a method to be the number of elements for which transfer is unnecessary divided by the number of elements actually transferred. Necessary transfer consists of the input and output of principal elements and the input of subordinate elements during the forward pass, and the input of each element during the backward pass. Unnecessary transfer occurs because strips and blocks are padded with zeroes, or non-subordinate elements are part of the I/O records containing subordinate elements. The expressions for the number of elements actually transferred come directly from Table 5-1.

For example, the first $\tilde{M}$ strips of the strip partitioning are

padded with zeroes since their columns are not of full length (see Figure 4-1). The SR and ST methods transfer these elements twice during the forward pass and once again during the backward pass. The total number of elements transferred is about $3N\tilde{M}$, while the unnecessary transfers total $3M^2/2$, so

$$FR_{SR} - FR_{ST} = (3M^2/2)/(3NM) = M/(2N).$$

This fragmentation is insignificant if $N \gg M$; thus we shall ignore the effects of transferring the extra triangle of elements. For the remaining methods, we derive FR by evaluating the costs for a full principal strip or block-column.

For the SS method, the number of elements transferred for a given principal strip is about $(\tilde{M}+3)KM$. There is unnecessary transfer in reading the $M^2/2$ extra elements along with the subordinate set. By using the fact that $K=M/\tilde{M}$, we simplify the ratio of these expressions to

$$FR_{SS} = \tilde{M}/(2\tilde{M}+6).$$

As $\tilde{M}$ varies from 14 to 4 (which uses from $.14M^2$ to $.5M^2$ primary memory), FR_{SS} ranges from .41 to .29, a high level of fragmentation.

For the block methods we derive the unnecessary I/O from the difference between actual I/O and necessary I/O. The amount of necessary I/O in the BC method is $3LM$ for the principal elements within a block-column plus $M^2/2$ for the subordinate elements. The number of elements actually transferred is $L^2\ (\tilde{M}^2/2 + 7\tilde{M}/2 + 3)$. Using $L=M/\tilde{M}$, the

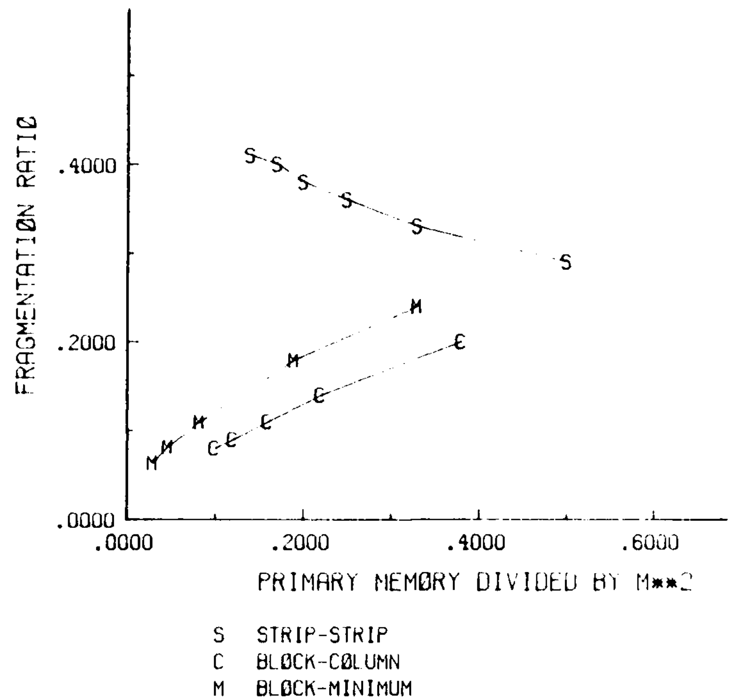


Figure 5-5: Fragmentation Ratios vs. Primary Memory Usage

difference between these expressions divided by the actual I/O reduces to

$$FR_{BC} = (\bar{M}+6)/(\bar{M}^2+7\bar{M}+6).$$

As $\bar{M}$ varies from 12 to 4 (which uses from $.1M^2$ to $.38M^2$ primary memory), FR_{BC} ranges from .08 to .20.

Finally, the expression for necessary I/O in the BM method is *slightly* more involved. In addition to the I/O of the BC method, there is also the input of blocks from the principal block-column when they contain elements of the first subordinate set. The lower-triangular block at the top of the column is read $\bar{M}$-1 times, and the total number of full block transfers is $(\bar{M}-1)\bar{M}/2$. Thus the total number of necessary transfers is

$$3LM + M^2/2 + (\bar{M}-1)L^2/2 + L^2(\bar{M}-1)\bar{M}/2.$$

The number actually transferred is $L^2(\bar{M}^2+3\bar{M}+3)$. From these expressions, the fragmentation ratio can be shown to be

$$FR_{BM} = (\bar{M}+7)/(2\bar{M}^2+6\bar{M}+6).$$

As $\bar{M}$ varies from 9 to 3 (using from $.04M^2$ to $.33M^2$ *primary memory*), FR_{BM} ranges from .07 to .24.

In Figure 5-5, we plot these fragmentation ratios against primary memory usage as expressed by the coefficient of M^2. In the SS method, fragmentation decreases as *strip size increases* because a given strip is reread fewer times as a subordinate strip and therefore its nonsubordinate elements are read fewer times. In contrast,

fragmentation increases with block size since a larger block size requires more padding. This explains why BM and BC have an advantage *over SS in the* τ *coefficient of I/O that narrows as the primary memory* usage increases (see Figure 5-4).

5.4 Memory Occupancy Costs

We now compare the memory occupancy costs of secondary storage methods with each other and with several in-core alternatives for solving linear systems. These results show that secondary storage methods may be of interest even when there is enough primary memory for in-core solution. Often, large problems are solved on mainframes where minimizing the dollar cost may be the most important consideration. In this situation, it can pay to use less memory at the cost of higher I/O and turn-around time.

Recall the model problem introduced in Chapter 1 with dimension $N=M^2$. The natural ordering of the grid points yields a symmetric positive definite coefficient matrix of bandwidth M. The factorization of this matrix in primary memory requires $O(M^4)$ time and $O(M^3)$ storage. *The memory occupancy cost is therefore* $O(M^7)$, which is by far the *largest asymptotic cost of solving the system. If the grid is ordered* by nested dissection and solved by sparse methods [12, 34] the work is reduced to $O(M^3)$ and the storage to $O(M^2 \log M)$, so memory occupancy is $O(M^5 \log M)$.

For the secondary storage methods, we derive separate components of memory occupancy associated with computation time and I/O time. In practice, the occupancy time depends on the extent to which I/O is overlapped with computation. But asymptotically, the occupancy cost depends on the dominant term regardless of overlap.

First, we use the expressions of Table 5-2, where the memory and I/O requirements per column were derived assuming that $\tilde{M}$ and $\hat{M}$ are held constant as M varies. Under this assumption, all secondary storage methods have the same asymptotic memory and I/O requirements, $O(M^2)$ and [illegible]$)\sigma+O(M^3)\tau$, respectively, so the I/O occupancy is $O(M^3)\sigma+O(M^5)\tau$. S[illegible]ll band factorization algorithms have $O(M^4)$ work, the co[illegible]tional occupancy is $O(M^6)$.

[illegible]here are more interesting results if we assume that K and L are held constant and use the requirements from Table 5-1. We summarize these computational and *I/O occupancy costs* in Table 5-3, including secondary storage methods and in-core alternatives. The results show that I/O occupancy is never more significant than computational occupancy, and that the lowest levels are achieved by the methods with the most I/O. Furthermore, the memory occupancy of the BM methods is as asymptotically efficient as any in-core method, direct or iterative! Any such method has a lower bound of O(N) work and storage, or $O(M^4)$ memory occupancy for the model problem.

Whether these decreases in occupancy charges would offset the I/O

Method	Primary Memory	Computational Occupancy	I/O occupancy
Band (In Memory)	$O(M^3)$	$O(M^7)$	0
Sparse (In Memory)	$O(M^2 \log M)$	$O(M^5 \log M)$	0
SR ST	$O(M^2)$	$O(M^6)$	$O(M^4)\ \sigma + O(M^5)\ \tau$
SS	$O(M)$	$O(M^5)$	$O(M^4)\ \sigma + O(M^5)\ \tau$
BC	$O(M)$	$O(M^5)$	$O(M^5)\ \sigma + O(M^5)\ \tau$
BM	constant	$O(M^4)$	$O(M^4)\ \sigma + O(M^4)\ \tau$

Table 5-3: Asymptotic Memory Occupancy Costs for the Model Problem

charges depends upon the relative cost of memory and I/O. This raises the issue of how memory and I/O charging rates should be determined. Secondary storage methods offer a tradeoff between the resources of a computer that could be used to exploit an imbalance in a charging algorithm. If I/O is relatively cheap, users would be encouraged to use as little memory as possible and perhaps overload the I/O system. Expensive I/O would favor the minimum-I/O methods or in core methods, which may result in an underused I/O resource. Paging systems usually handle this problem by using I/O only when absolutely necessary.

However, their efficiency varies greatly between programs and at different levels of memory usage, and is generally beyond the direct control of the user. Secondary storage codes allow this memory-I/O tradeoff to be efficiently controlled and perhaps exploited in ways that no other methods offer.

In Chapter 6, we shall develop a more specific analysis of memory occupancy costs in the case where I/O is overlapped with computation.

CHAPTER 6

Parallel Execution of Computation and I/O

6.1 Introduction

We now consider how secondary storage methods could exploit a capability for performing I/O synchronized in parallel with computation. This capability exists in architectures where I/O and computation can be performed by separate asynchronous or synchronous processors that use the same primary memory. Our objective is to use this capability to minimize the time that the computation processor is idle while waiting for I/O. For several of the methods, we develop schemes for overlapping all I/O during the factorization, with the exception of an initial read and final write. We also examine the effect that overlapped I/O has on turn-around time and memory occupancy.

Not all numerical algorithms are well suited for the parallel execution of I/O. In particular, if input of a record is to be carried out before that record is actually needed, then the sequence of input operations must not be affected by the actual values being computed. This is a property held by all of the nonpivoting secondary storage

methods introduced in Chapter 4. It can also be true of a pivoting algorithm, but only if the elements involved in the pivot search and row exchange are in primary memory. For all such algorithms, the order in which records must be input and output can be predetermined, and can therefore be scheduled in parallel with computation on other records.

Our approach for determining the requirements and capabilities of parallel I/O is to specify a synchronization of the required computation and I/O of a given secondary storage method. That is, we schedule I/O events in parallel with computational events so as to meet the precedence requirements. Using the models of Chapter 5 to express the time required for these events, we derive conditions under which each I/O event will be completed before its simultaneous computational event. From these conditions and the storage requirements of the methods, we determine the amount of primary memory needed to overlap virtually all such I/O in the factorization. Under these conditions, the turn-around time of secondary storage methods is nearly the same as solving the system totally within primary memory.

A property of a secondary storage method that reflects the ease of developing parallel synchronizations is the local work-I/O ratio, the computational cost divided by the I/O cost during a given segment of a program's execution. Secondary storage methods whose work-I/O ratio is relatively constant throughout their execution tend to be easy to synchronize, while those whose ratio varies greatly are less easily

synchronized. Put another way, it is easier to overlap I/O with computation when the rate of data flow matches the rate of work.

We say that a computation is compute-bound during a certain time period if the CPU is never idle, and thus the turn-around time is bounded by the computational events required. The turn-around time of a compute-bound scheme cannot be reduced further without reverting to a different algorithm with lower computational requirements.

In Section 2, we give examples of computing environments that allow for parallel I/O and computation. Included are peripheral array processors in various configurations, and FORTRAN run-time systems that allow overlapped I/O.

In Section 3, we introduce synchronization with the SK or ST methods, showing I/O and memory management schemes. These schemes include both double-buffering with a single I/O channel and triple-buffering with two channels.

In Section 4, we derive expressions for the minimum amounts of memory needed for compute-bound SK and ST factorization. These conditions depend upon the bandwidth as well as the rates of computation and I/O. We show that the methods are nearly always compute-bound for typical machine parameters. Furthermore, if the start-up time σ is negligible, then the conditions for compute-boundedness depend only on the bandwidth and are independent of strip size.

In Section 5, we present a synchronization of the BM method and derive conditions for compute-bound execution. These conditions give an *upper bound on the amount of primary memory needed to totally overlap* I/O with computation that is independent of bandwidth! That is, we show that a fixed amount of memory is sufficient to compute the factorization for any size system with nearly the same turn-around time as an in-core factorization.

We demonstrate the difficulty or impossibility of achieving a compute-bound back-solve in Section 6. In Section 7, we consider the implications of compute-boundness on minimizing turn-around time, and, in Section 8, we show the effects of parallel execution of I/O and computation on memory occupancy costs.

A synchronous model does not necessarily reflect the performance of parallel computation and I/O, which is asynchronous by nature. Rather, the synchronization of events indicates the capability of the secondary storage algorithm to achieve compute-boundedness within a certain amount *of primary memory*. This may also have implications on how a computing environment can be designed to efficiently use all its resources.

6.2 Hardware and Software Allowing Parallel I/O

The capability for parallel execution of I/O and computation exists in several types of computing environment. It is especially important in those *configurations* that require the extensive use of a memory hierarchy. This is often true for very high-speed machines that are suitable for doing large-scale numerical computations.

One such configuration is a *peripheral array processor* (AP) coupled in some manner to a host machine [2]. The AP can be used to perform floating-point computations at a much faster rate than the host. The high speed of an AP requires very fast memory in order that the processor not be memory-bound, so memory can be the dominant cost of an AP unless data channels allow the *effective use of a limited amount of* memory. Therefore, the design and use of links between the host, the AP, and secondary storage devices is essential to their efficient use and cost-effectiveness.

Thus, AP's generally have good capabilities for carrying out concurrent I/O and computation. This is accomplished with special-purpose processors within the AP for handling the control and addressing of transfers. Other features that facilitate parallel I/O are multiple memory busses so that the processors do not steal memory cycles from each other, and a control processor to handle interrupts and

(A) (Disk) <--> [Host] <=> [AP]

(B) [Host] <=> [AP] <=> (Disk)

(C) [Host] <=> [Dual-Port Bulk Memory] <=> [AP]

(D) (Disk) <=> [Host | AP]

Figure 6-1: Configurations of Host, AP, and Secondary Storage

---

synchronization between processors without the need for host intervention.

The network of data channels and memories can take various forms, some of which are illustrated in Figure 6-1. In example (A), the host controls disk I/O and passes data to the AP through a peripheral interface channel. This configuration is the least desirable from the standpoint of efficiency and convenience because there are two steps to a transfer between AP and disk. An AP may be able to carry out secondary storage transfers itself directly to and from disk storage (B)

or bulk memory (C), and in either case it is possible to have dual port access by the host as well, as shown with bulk memory in (C). Finally, it is possible to have a host and AP interfaced through shared memory. This allows the most convenient control over disk I/O, which can be carried out by the host directly into AP memory using standard FORTRAN reads and writes.

Other machines that allow some form of overlapped I/O are computers designed for large scale numerical computations. For example, the CDC 6600/7600 FORTRAN allows I/O to be overlapped through BUFFER IN and BUFFER OUT commands [9]. These commands are essentially the same as READ or WRITE except that the the program continues execution after the *transfer of a record is initiated. Before the program references an* element of that record or performs more I/O with the same file, there must be a call to the function UNIT, which delays program execution until the transfer in progress is completed. Thus, simple sequential I/O can be overlapped, but the responsibility for avoiding memory conflicts is left to the user. Such a mechanism exists in similar forms on other machines, including the CRAY-1 [11, 36].

As we discussed in Chapter 3, it would also be possible and desirable for a paging system to have the same capability. *If a page* transfer could be initiated before that page is actually used, then both the transfer and subsequent computation could be carried out concurrently.

The storage and I/O schemes in the following sections are not designed with any such specific hardware or software features in mind. However, the strip methods with overlapped-I/O of the next section could be easily implemented with a simple mechanism such as that described for FTN. The more complex overlapped-I/O scheme for the BM method would require more elaborate I/O queuing and memory management schemes. Therefore, it may be of more interest for its theoretical implications in the design of new machines than as a practical method for carrying out parallel I/O on existing machines.

6.3 Synchronization and Storage Schemes for SR and ST Methods

In order to demonstrate the synchronization of parallel computational and I/O events, we start with the SR and ST minimal-I/O methods. These methods are easiest to synchronize because they have constant work-I/O ratios for most of their execution time. That is, after the first $\tilde{M}$ non-full strips, the factorization of each strip requires about $KM^2/2$ multiplies, along with the input and output of that strip.

The approach we shall use to overlap I/O in the SR and ST methods resembles the pipelining of the stages of an arithmetic operation as used in various scientific computers [20]. In this case, each strip is involved in three stages: input, computation, and output. These stages can be overlapped between successive strips in a kind of I/O pipeline.

(A) Sequential Execution: time (-step-)

```
| Input |    |Compute|    | Output|    | Input |    |Compute|    | Output|
| Strip | -> | Strip | => | Strip | => | Strip | => | Strip | -> | Strip | ->
|   1   |    |   1   |    |   1   |    |   2   |    |   2   |    |   2   |
```

(B) Pipelined Execution for the ST Method

```
| Input |    | Input |    | Input |
| Strip | => | Strip | => | Strip | => . . .
|   1   |    |   2   |    |   3   |

           |Compute|    |Compute|    |Compute|
           | Strip | => | Strip | => | Strip | => . . .
           |   1   |    |   2   |    |   3   |

                        | Output|    | Output|    | Output|
                        | Strip | => | Strip | => | Strip | => . . .
                        |   1   |    |   2   |    |   3   |
```

(C) Pipelined Execution for SR without memory conflicts, $\tilde{M}$-2:

```
| Input |    | Input |    | Input |    | Input |    | Input |
| Strip | => | Strip | => | Strip | => | Strip | => | Strip | => . . .
|   1   |    |   2   |    |   3   |    |   4   |    |   5   |

           |Compute|    |Compute|    |Compute|    |Compute|    |Compute|
           | Strip | => | Strip | => | Strip | => | Strip | => | Strip | ->
           |   1   |    |   2   |    |   3   |    |   4   |    |   5   |

                                                  | Output|    | Output|
                                                  | Strip | => | Strip | ->
                                                  |   1   |    |   1   |
```

Figure 6-2: Pipelining of I/O in the SR and ST Methods

Our use of pipeline terminology in connection with I/O should not be confused with conventional arithmetic pipelining.

The SR and ST methods contain the sequence of I/O and computational *events pictured in Figure 6-2(A). Parallel pipelined I/O with the ST* method is achieved by overlapping the computation of a strip with the input of the succeeding strip and the output of the preceding strip. Shown in Figure 6-2(B), this synchronization assumes that a "time step" equals both the computation and the transfer time for a strip. The conditions under which this is true will be explored in the next section. It also assumes that there are two independent I/O channels that can simultaneously perform input and output. We later consider the changes necessary for only one channel.

In practice, the synchronization of the SR method may have to be slightly different because the elements of U for a given strip are subordinate elements during the factorization of the next $\tilde{N}$ strips. *Even though the subordinate elements are not modified* in the inner-product algorithm, it may not be possible to output them and compute with them simultaneously because of memory contention or buffer protection mechanisms. This conflict may be avoided by delaying the output of a strip for $\tilde{N}$ time steps until it no longer contains subordinate elements. The pipeline of 6-2(C) shows this output delay for the case of $\tilde{N}$=2.

1. INPUT Strip 1 ;
2. INPUT Strip 2 and COMPUTE Strip 1 ;

   FOR J = 2 TO $\tilde{N}$-1 DO

3. [INPUT Strip J+1, COMPUTE Strip J, and OUTPUT Strip J-1] ;
4. COMPUTE Strip $\tilde{N}$ and OUTPUT Strip $\tilde{N}$-1 ;
5. OUTPUT Strip $\tilde{N}$

Algorithm 6-1: ST Method with Parallel Computation and 2-Channel I/O

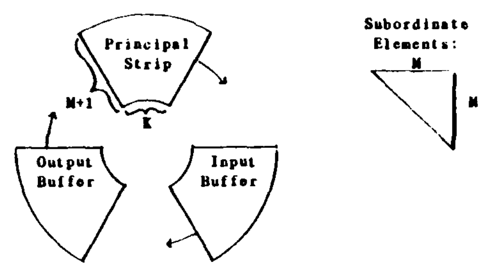


Figure 6-3: ST 2-Channel Buffering Scheme

The parallel execution of I/O and computation requires that more memory be used than the amounts specified in Chapters 4 and 5. This memory is needed as buffer space for the strips being input and output. The synchronization and buffering schemes for the ST method are shown in

Algorithm 6-1 and Figure 6-3, where the operations at each numbered line are carried out simultaneously. For the two-channel case, this requires three strips of memory that circularly rotate between input buffering, computation, and output buffering. The subordinate elements may be stored in a separate array which is not involved in any I/O. Thus, the events are synchronized as in the pipeline scheme of Figure 6-2(B).

With the SR method, subordinate elements are not stored separately so the buffering scheme is a bit more complex. Figure 6-4 illustrates how primary memory can be allocated for the various strips involved in I/O or computation at any given time. A given strip in primary memory is used first as an input buffer, then as a principal strip for one time step, as a subordinate strip for $\tilde{N}$ steps, and finally as an output buffer, before being recycled through this sequence for another strip. A similar storage and I/O scheme was used in [15] to overlap I/O using the BUFFER IN and BUFFER OUT commands of CDC 7600 FORTRAN.

Notice that we do not output the final $\tilde{N}+3$ strips in Algorithm 6-2. We may not need to write them out since they are used in the first steps of the backward pass. The output file must be completed only if additional right-hand sides are to be solved later. We shall henceforth assume that output of a strip is not carried out unless its *memory space* is needed later as an input buffer. Thus, at the end of the factorization, the entire memory space is filled with strips which were never output and can be used as the first strips in the back-solve.

1. INPUT Strip 1 ;

   FOR J = 1 TO $\tilde{N}$+1 DO

2. [INPUT Strip J+1 and COMPUTE Strip J] ;

   FOR J = $\tilde{N}$+2 TO $\tilde{N}$-2 DO

3. [INPUT Strip J+1, COMPUTE Strip J, and OUTPUT Strip J-1-$\tilde{N}$] ;

4. INPUT Strip $\tilde{N}$ and COMPUTE Strip $\tilde{N}$-1 ;

4. COMPUTE Strip $\tilde{N}$

Algorithm 6-2: SR Method with Parallel Computation and 2-Channel I/O

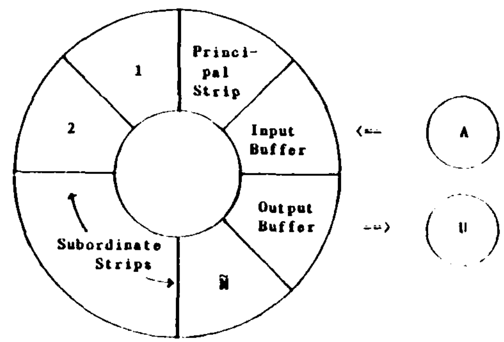


Figure 6-4: SR 2-Channel Buffering Scheme

1. INPUT Strip 1 ;

   FOR J = 1 TO $\tilde{M}$+1 DO

2. [INPUT Strip J+1 and COMPUTE Strip J] ;

   FOR J = $\tilde{M}$+2 TO $\tilde{N}$-1 DO

3. [OUTPUT Strip J-1-$\tilde{M}$ then INPUT Strip J+1 and COMPUTE Strip J] ;

4. COMPUTE Strip $\tilde{N}$

Algorithm 6-3: SM Method with Parallel Computation and 1-Channel I/O

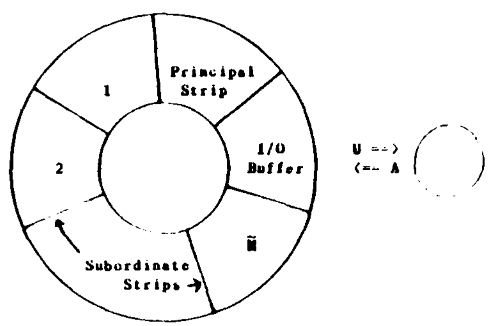


Figure 6-5: SM 1-Channel Buffering Scheme

We have assumed throughout these synchronizations that the input and output of two different strips can be carried out simultaneously through two independent I/O channels. If only one such data channel exists, two changes occur. First, the I/O will take twice as long without overlap of input with output. Second, only one buffer is needed in the circular memory scheme. After a strip has been output to secondary storage, the same buffer is used to input the next strip. We incorporate these changes into Algorithm 6-3 and in Figure 6-5. Similarly, the ST method would require only one buffer with one data channel.

6.4 Conditions for Compute-Bound Strip Factorization

We now derive conditions under which the synchronized factorizations of Section 3 are completely compute-bound after a *start-up period*. Clearly, the CPU must be idle during the input of the first strip. Furthermore, the first $\tilde{M}$ strips contain short columns that require less computation than full strips. This is a low order effect, so let us consider the conditions under which all I/O can be overlapped with computation after these initial strips.

The first case is that of two I/O channels performing concurrent input and output. The I/O in Line 3 of Algorithms 6-1 and 6-2 takes about $\sigma+\tau KM$ time to finish. The factorization for one strip takes about $\mu KM^2/2$ time. Therefore, the I/O finishes first and the loops are

compute-bound if

$$\mu KM^2/2 > \sigma + \tau KM \quad \text{or} \quad K > 2\sigma/(\mu M^2 - 2\tau M). \qquad (6.1)$$

Inequality (6.1) is never satisfied if $M \leq 2\tau/\mu$, but μ and τ are generally of similar enough magnitude that this is not the case. We define $K_{cb2}=2\sigma/(\mu M^2 \; 2\tau M)$, that is, *the minimum value of K for which the two channel case is compute-bound.*

With only one channel, the I/O takes twice as long, so the equivalent conditions are

$$K > 4\sigma/(\mu M^2 - 4\tau M), \qquad (6.2)$$

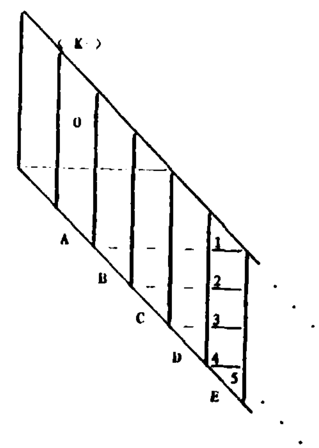
and $M \leq 4\tau/\mu$. We define $K_{cb1}=4\sigma/(\mu M^2 - 4\tau M)$, the minimum strip size for which the SR or ST method is compute-bound.

If we ignore σ, then the compute-bound overlap conditions arising from (6.1) and (6.2) *are independent of K, but require that*

$$M > 2\tau/\mu \quad \text{or} \quad M > 4\tau/\mu \; ,$$

respectively. This is not surprising in view of the fact that the τ *component of the SR/ST I/O function is independent of K.* Thus, increasing the strip-size only serves to reduce the effect of σ.

Experimental timings with the DEC System 2060, to be presented in Chapter 7, show that sequential FORTRAN I/O achieves values of μ=.01 ms., σ-21 ms., and τ-.02 ms. For these rates of computation and I/O, the SR and ST methods is always compute-bound with two channels if $M > 67$, and with one channel if $M > 96$. That is, K_{cb2} and K_{cb1} equal 1 under these conditions and are small under most other practical cases.



I/O	COMPUTATION	Number of Multiplies
Input E		
Input A		
	Compute Region 1	$K^3/6 + K^2/2 + K/3$
Input B		
	Compute Region 2	$K^3 + K^2$
Input C		
	Compute Region 3	$2K^3 + K^2$
Input D		
	Compute Region 4	$3K^3 + K^2$
	Compute Region 5	$11K^3/6 + K^2$
Output E		

Figure 6-6: *I/O vs. Computation in the SS Method*

Using these conditions, we can derive the amounts of primary memory that are sufficient for compute bound factorization with the SK and SI methods. These results will be summarized at the end of Section 5.

The SS method, in which subordinate blocks are kept in secondary storage, is not as suitable for overlapping I/O because its work-I/O ratio varies greatly through the *factorization* of a principal strip. This is *illustrated in* Figure 6-6, which shows the sequence of I/O events and computational events with multiplication counts. Since each region requires a different amount of computation, and in some cases a *different amount of I/O*, the work-I/O ratio constantly changes. Any scheme *for* overlapping the *I/O* would be awkward because of the incompatible rates of data flow and computation.

6.5 Synchronization and Compute-Boundedness with the BM Method

Next, we show that a compute-bound synchronization is possible even with the highest level of I/O in the BM method (see Algorithm 4-4), which performs block factorization using three L by L blocks of primary memory. We use *the multiplication* counts for the various block operators from Table 2-2 *to show that the overlap of I/O is possible* using just five blocks of primary memory and one I/O channel.

Suppose that L_{cb} is the smallest block size so that one block transfer can be performed in the time needed for $L^3/6$ multiplies, i.e.,

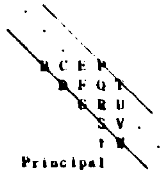
$$\sigma+\tau L_{cb}^2 = \mu L_{cb}^3/6. \qquad (6.3)$$

We assume that L is a factor of M, so that there is no band pad. In this case, each of the block operators requires either L^3, $L^3/2$, or $L^3/6$ multiplies, so if $L \geq L_{cb}$, then a block operator requires enough time to *totally overlap 6, 3, or 1 block transfers, respectively.*

Figure 6-7 shows the block operators for the factorization of one *block-column by the BM method, scaled to those time requirements. That* is, opposite each operator are slots for the number of I/O events that could be overlapped if $L \geq L_{cb}$. We show the case of $\hat{M}=3$, but larger block bandwidths with the same block size are even easier to synchronize *because there is a higher proportion of full blocks, and therefore a* higher work-I/O ratio. The synchronization shown is the result of scheduling input and output by the following simple rules:

1. Each output operation occurs in the first slot after a modified block is no longer used in a current block operator;
2. The input operations are then scheduled in the slot(s) immediately before a block not already in primary memory is needed.

Those operations involving blocks from the preceding or succeeding block-column are in parentheses. The block set is shown only for those *steps in which it changes, and a semicolon separates those blocks* involved in the present operator from those being transferred or stored for future use. This schedule demonstrates that if $L \geq L_{cb}$, this entire sequence is compute-bound using a block set no greater than five.



P and T are lower triangular, and B,D,G,S, and W are symmetric.

Principal Block-Column

Block Operator	I/O Event	Block Set
$G = G^{1/2}$	(C in)	G;(P,B,Q,C)
$P = P/B$	(G out)	P,B;Q,C,(G)
		Q,C,P
$Q = Q-C^TP$		
	D in	Q,C,P;D
	P out	Q,D;P
$Q = Q/D$	R in	Q,D;R
	F in	Q,D;R,F
		R,F,Q
$R = R-F^TQ$		
	E in	R,F,Q;E
	P in	R,F,Q;E,P

Continued in the next column

Block Operator	I/O Event	Block Set
	Q out	R,E,P;Q
$R = R-E^TP$		R,E,P
	G in	R,E,P;G
		R,G
$R = R/G$		
	S in	R,G;S
		S,R
$S = S-R^TR$	Q in	S,R;Q
	P in	S,R;Q,P
	R out	S,Q;R,P
$S = S-Q^TQ$	(T in)	S,Q;P,(T)
	(D in)	S,Q;P,(T,D)
$S = S-P^TP$	(U in)	S,P;(T,D,U)
$S = S^{1/2}$	(F in)	S;(T,D,U,F)
$T = T/D$	(S out)	T,D;U,F,(S)
	⋮	

Figure 6-7: Synchronization of BM Method for a Block-Column

P is lower triangular, B,D,G and S are symmetric.

Block Operator	I/O	Block Set
	B in	B
$B = B^{1/2}$	C in	B;C
		C,B
$C = C/B$		
	D in	C,B;D
	B out	D,C;B
$D = D-C^TC$	B in	D,C;B
	E in	D,C;B,E
$D = D^{1/2}$	C out	D;C,B,E
	D out	E,B;D
$E = E/B$	F in	E,B;F
	C in	E,B;F,C

Continued in the next column

Block Operator	I/O	Block Set
$F = F-C^TE$		
	D in	F,C,E;D
	E out	F,D,E
$F = F/D$		F,D
	G in	F,D;G
		G,F
$G = G-F^TF$	E in	G,F;E
	(P in)	G,F;E,(P)
	F out	G,E;F,(P)
$G = G-E^TE$	(B in)	G,E;(P,B)
	(Q in)	G,E;(P,B,Q)
$G = G^{1/2}$	(C in)	G;(P,B,Q,C)
$P = P/B$	(G out)	P,B;Q,C,(G)

Figure 6-8: Synchronization of First $\tilde{W}$ Block Columns

What about the synchronization at the beginning of the factorization, involving the first M short block-columns? Here, there are no lower-triangular blocks from the edge of the band and therefore there is relatively more computation with which to overlap I/O. Thus, we apply a slightly different synchronization, shown in Figure 6-8. Notice that this scheme is compute-bound as soon as the first block has been input, and that all the advance input needed to lead into the main part of the algorithm has been carried out at the end of the third block-column. These schedules prove that if $L \geq L_{cb}$, then the entire factorization is compute bound after the input of the first block.

Being the root of a cubic polynomial, L_{cb} has no explicit representation for the general case, but the following special cases are of interest. If $\sigma \gg \tau$, we ignore τ, giving a compute-bound condition of

$$L > L_{cb\sigma} = (6\sigma/\mu)^{1/3}.$$

On the other hand, if σ is negligible, we obtain

$$L > L_{cb\tau} = 6\tau/\mu.$$

Finally, if we use the empirically determined values from Chapter 7 for random access I/O with the DEC System 2060 (μ=.01ms., σ=41 ms. and τ=.04 ms.), then we can determine a value for L_{cb} from (6.3). This value is about 40. Therefore, the amount of memory necessary for the BM method to be compute bound with this system is only 8000 words.

Those synchronizations are not intended to be algorithms, but serve as a sample schedule showing that compute-bound execution is possible.

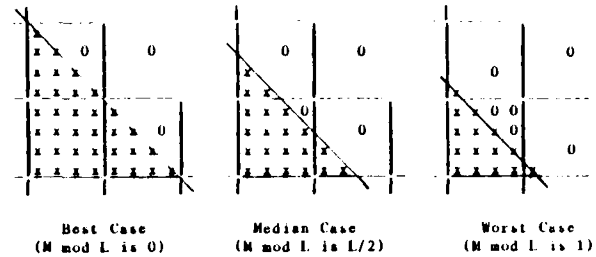


Figure 6-9: Best to Worst Cases of Band Padding

This schedule is a special case because of the assumption, implicit in the multiplication counts, that there is no band pad. In fact, this is the best case with respect to work-I/O ratio. Figure 6-9 shows the median and worst cases of band padding. For these cases, the I/O requirements are identical but the computation involved with these blocks is less. As a result, the work-I/O ratio varies more, and a compute-bound synchronization is not as easy to achieve. By the same process as is described above for the given schedules, the average and worst cases can be synchronized under condition (6.3) if the block set size is *increased* to 7 and 9, respectively.

While compute bound synchronization of the SK and ST methods is more straightforward, the implications of this compute-bound block synchronization are more significant. Condition (6.3) and the block set size combine to make an upper bound on the primary memory necessary for

Method	Case	Compute-Bound Record Size	Minimum Compute-Bound Memory Requirement
SR	1 Channel	$K > \frac{4\sigma}{\mu M^2 - 4\tau M}$	$M^2 + \frac{8\sigma}{\mu M - 4\tau}$
	2 Channels	$K > \frac{2\sigma}{\mu M^2 - 2\tau M}$	$M^2 + \frac{6\sigma}{\mu M - 2\tau}$
	1 Channel (σ=0)	$M > 4\tau/\mu$	$M^2 + 2M$
	2 Channels (σ=0)	$M > 2\tau/\mu$	$M^2 + 3M$
ST	All Cases	Same as SR	Same as SR except $M^2/2$ instead of M^2
BM	$\sigma >> \tau$	$L > (6\sigma/\mu)^{1/3}$	$5\ (6\sigma/\mu)^{2/3}$
	σ neglibible	$L > 6\tau/\mu$	$180\ (\tau/\mu)^2$

Table 6-1: Summary of Compute-Bound Requirements

compute-boundedness that is <u>independent</u> of the size of the problem. That is, within a constant amount of primary memory, all of the I/O in the block band Cholesky algorithm can be totally overlapped with computation except input of the first block and output of the last block. This upper bound is $5L_{cb}^2$, where L_{cb} depends only upon the system performance parameters. The only demands on the size of the problem are

that the bandwidth be large enough so that $N \geq 3$ is satisfied, that is, $M \geq 3L_{cb}$.

In Table 6-1, we summarize the conditions on K and L for the strip and block synchronizations to be compute-bound. We also show the minimum amounts of primary memory required for computation and buffering when these conditions are met.

6.6 The Barrier to a Compute-Bound Back-Solve

In the previous sections, we have shown how nearly all of the I/O in the SR, ST, and BM factorizations can be overlapped with computation. We now consider the same problem for the strip or block back-solve.

The analysis of the back-solve is quite simple since each element of U is used just once for each right-hand side during the back-solve. Thus, for strip methods, each strip is used for about KM multiplies, and the condition for compute-boundedness is

$$\mu KM > \sigma + \tau KM, \text{ or}$$

$$K > \sigma / M(\mu - \tau).$$

For block partitioning, the compute-bound condition for a full block is

$$\mu L^2 > \sigma + \tau L^2, \text{ or}$$

$$L > \sqrt{\sigma / (\mu - \tau)}\ .$$

Neither of these conditions is possible unless $\mu > \tau$, that is, unless elements can be transferred from secondary storage faster than they can

be multiplied. This is generally impossible with most forms of secondary storage.

The only situation that allows a compute-bound back-solve is if there are multiple right-hand sides to be simultaneously solved. For R right-hand sides, each element of U is involved in R multiplies, and a large enough R would create enough computation with which to overlap the I/O.

The back-solve problem, in which I/O dominates computation, has been pointed out in the literature [20, 26]. Although the I/O takes a greater proportion of total time in the backward pass, the empirical results of Chapter 7 will show that the turn-around time is still dominated by the forward pass.

6.7 Analysis of Turn-Around Time

In this chapter, we have shown the conditions under which the SR, ST and BM factorizations are compute-bound with parallel execution of I/O and computation. The back-solve was found to be I/O-bound in most circumstances. We now derive expressions for the turn-around time of solving a linear system with these methods for the compute-bound and I/O-bound cases. These expressions are used in the next section to analyze the memory occupancy costs and determine how they are minimized.

In deriving these expressions, we make certain simplifying

assumptions. As done in previous chapters, we approximate $M+1$ with M wherever possible. Although there is less computation involved with the first few non-full strips than with the rest, we choose to ignore this low order effect and assume that the entire factorization is either compute-bound or I/O-bound. We do not include any output of strips after the factorization is finished, since these strips are the first to be used in the back-solve. Finally, since I/O dominates the low order computation in the back-solve, we assume that there is no overlap and we ignore the computation.

We first consider the turn-around time required for the SR or ST synchronization schemes. For a compute-bound factorization with either 1 or 2 channels, all transfers except the input of the the first strip are completely overlapped with computation. The computation involves $NM^2/2$ multiplies, so the turn-around time is

$$(\sigma+\tau KM) + \mu NM^2/2. \tag{6.4}$$

If the *factorization* is I/O-bound, then the turn-around time with 2 channels is

$$(\tilde{N}+1)(\sigma+\tau KM), \tag{6.5}$$

and with 1 channel,

$$(2\tilde{N}+1)(\sigma+\tau KM). \tag{6.6}$$

Notice that (6.4) is minimized by choosing K as small as possible. This means that increasing the strip size beyond the minimum compute-bound requirement only slows down the factorization since there is a

longer wait for the initial strip to be *input*. The *remaining* expressions are minimized by choosing K as large as possible. Since the largest I/O-bound strip size and the smallest compute-bound strip size are essentially the same, a choice of $K=K_{cb2}$ or K_{cb1} minimizes the turn-around time for the factorization. When σ is negligible, the choice of K is irrelevant.

Suppose we also include the I/O-bound back-solve in these expressions. If the first strip is already in primary memory, the turn-around time for all cases is about

$$(\tilde{N}-1)(\sigma+\tau KM). \qquad (6.7)$$

Thus the total turn-around times for the forward-backward pass corresponding to (6.4), (6.5), and (6.6) are

$$\tilde{N}(\sigma+\tau KM) + \mu N M^2/2 \;, \qquad (6.8)$$

$$2\tilde{N}(\sigma+\tau KM) \;, \text{ and} \qquad (6.9)$$

$$3\tilde{N}(\sigma+\tau KM) \;, \qquad (6.10)$$

respectively. In these cases, a larger choice of strip size always speeds up the overall turn around time because of the I/O-bound back-solve, but the effect is 2 or 3 times greater when the factorization is I/O bound.

For the block factorization, the same type of analysis is not as simple because of the non-uniform synchronization of I/O and computation. If $L<L_{cb}$, then there are several points in Figure 6-7 where the computation would wait for I/O to be completed, while most of

the *synchronization would still be* compute-bound. Thus the turn around time for this case is not easily expressed. If $L\geq L_{cb}$ then the turn-around time per block-column is

$$3\bar{M}^2_{cb}\ (\sigma+\tau L^2_{cb}), \qquad (6.11)$$

where $\bar{M}_{cb}=M/L_{cb}$. This uses the fact that there are $3\bar{M}^2$ slots per full block-column in Figure 6-7. Choosing block sizes larger than L_{cb} only increases the turn-around time of the factorization, since it takes longer for the initial block to be input.

A larger block size would improve the turn-around time of the I/O-bound back-solve, which is about

$$(1+\bar{M})\ (\sigma+\tau L^2).$$

per block-column. However, (6.11) is $3\bar{M}$ times larger, so the back-solve takes a small fraction of the total solution time even though it is I/O-bound.

6.8 The Effect of Parallel I/O on Memory Occupancy Costs

In Chapter 5, we showed that secondary storage methods have low asymptotic memory occupancy costs regardless of whether the I/O is overlapped with computation. These costs were lower than in-core factorization algorithms and even comparable with those of in-core iterative methods. In this section, we compare the memory occupancy for *various buffering and I/O strategies for carrying out parallel I/O and*

computation with the SR method.[*] There are two ways in which memory is used to decrease turn around time: for larger records that decrease total I/O time; and *for buffering so that transfers* and computation are in parallel. However, the increased use of memory and the decreased turn-around time have opposite effects on the memory occupancy costs. Under what conditions does the increased use of memory and the resulting decrease in occupancy time cause memory occupancy costs to fall, and when do they rise? This question, which was discussed with respect to overlapped I/O on the Cray-1 in [36, 11], is the motivation for this section. Let us separately consider the two ways in which memory use is increased.

First, we present expressions for the memory occupancy costs of the SR method for the following cases:

[*] The extension of results to the *ST method is a simple exercise.*

I. No overlap of computation and I/O;

II(A). Compute-bound overlap of computation and I/O through 1 channel;
II(B). I/O-bound overlap of computation and I/O through 1 channel;

III(A). Compute-bound overlap of computation and I/O through 2 channels;
III(B). I/O-bound overlap of computation and I/O through 2 channels;

The memory occupancy cost is the product of turn-around time and primary memory usage. For Case I, the turn-around time for the factorization is about

$$2\tilde{N}(\sigma+\tau KM) + \mu NM^2/2, \tag{6.12}$$

and the *primary memory requirement* is $KM+M^2$ words. For Cases II(A) and II(B), the turn-around times are given by (6.4) and (6.5), respectively, using $2KM+M^2$ words. For Cases III(A) and III(B), the turn-around times are given by (6.4) and (6.6), respectively, using $3KM+M^2$ words. We shall assume that there is no overlap and no buffering in the back-solve, since we choose to ignore its low order computational costs. Therefore, its turn-around time is given by (6.7) using KM words of primary memory for all cases.

We present the memory occupancy costs for the forward and backward passes together in Table 6-2. These are simply the products of the turn-around times and memory requirements for the appropriate cases, with a back solve occupancy added in, as approximated by $\tilde{N}(\sigma+\tau KM)(KM)$.

To simplify the expressions, we assume that K evenly divides N; thus $K\tilde{N}=N$.

Now, consider minimizing memory occupancy through the choice of strip size. By differentiating the expressions in Table 6-2 with respect to K for each case, we obtain the values presented in Table 6-3. The result for each compute-bound case is a negative value of K. This indicates that memory occupancy is minimized by choosing a value for K that results in a totally I/O-bound computation. That is to say, the greater turn-around time due to I/O-bound execution is generally offset in memory occupancy costs by the smaller memory usage. Whether the savings in memory occupancy costs are greater than the increased I/O costs depends on the relative charges for occupancy and I/O.

Case	Memory Occupancy Costs, Forward + Backward
I	$\mu(NM^4+KNM^3)/2 + \sigma(2\tilde{N}M^2+3NM) + \tau(2NM^3+3KNM^2)$
II(A)	$\mu(NM^4+2KNM^3)/2 + \sigma(NM+M^2+2KM) + \tau(KNM^2+KM^3+2K^2M^2)$
II(B)	$\sigma(4NM+2\tilde{N}M^2+NM) + \tau(3KNM^2+2NM^3)$
III(A)	$\mu(NM^4+3KNM^3)/2 + \sigma(NM+M^2+3KM) + \tau(KNM^2+KM^3+3K^2M^2)$
III(B)	$\sigma(3NM+\tilde{N}M^2+NM) + \tau(4KNM^2+NM^3)$

Table 6-2: Memory Occupancy for SR Overlap/Buffering Schemes

Case	Strip Size Minimizing Memory Occupancy
I	$K = \sqrt{\frac{4\sigma}{M\mu + 6\tau}}$
II(A)	$K < 0$
II(B)	$K = \sqrt{\frac{2\sigma}{3\tau}}$
III(A)	$K < 0$
III(B)	$K = \sqrt{\frac{\sigma}{4\tau}}$

Table 6-3: Strip Sizes Minimizing Memory Occupancy

Finally, there is the question of whether the memory occupancy costs are reduced by using extra memory for overlapped I/O. We address this question by constraining the three cases in Table 6-2 to operate within equal amounts of memory, and comparing their occupancy costs. Under this constraint, the strip size of Case I can be twice as large as that of Case II, and three times that of Case III. Since the cases all use the same memory, we can compare memory occupancy by comparing the turn-around times given by Equations (6.8), (6.9), (6.10), and (6.12).

We find that there are rare circumstances when Case I is best, and the memory occupancy costs are minimized by not overlapping any I/O. Case I has less turn-around time than Case II(B) if the strip size for Case II(B) is no more than $3\sigma/(M^2\mu+2M\tau)$. Case I wins over III(B) if the strip size for III(B) is no more than $2\sigma/M^2\mu$. That is to say, a small enough strip size can cause the I/O-bound cases to take more time than the no-overlap case within the same amount of memory.

In comparing one-channel versus two-channel synchronizations, we find that memory occupancy is always less for Case III unless both methods are compute-bound. In that case, the one-channel synchronization has quicker turn-around because its larger strip size reduces the I/O-bound back-solve time. In other words, the use of separate I/O channels for overlapping input and output with each other does not reduce memory occupancy or turn-around time unless there is not enough primary memory for compute-bound execution with one channel.

The results of this chapter have significant implications on the storage requirement of factorization algorithms. With parallel execution of computation and I/O, there is an upper bound of $5L_{cb}^2$ on the amount of primary memory needed to keep a processor busy during the factorization of a symmetric, positive definite banded matrix. Using more primary memory, and even solving the system totally in primary memory, can only improve turn-around time slightly while adding substantial increases in primary memory costs.

CHAPTER 7

Implementation and Performance of the Methods

7.1 Introduction

In this chapter, we discuss the issues involved in implementing the secondary storage methods and report on the performance of these methods. The characteristics of secondary storage methods demand that special attention be paid to implementation. In particular, the heavy dependence on I/O requires that transfers be carried out in as efficient a manner as possible. In this section, we describe the design features of a package of routines called BESS, for Band Elimination using Secondary Storage. BESS includes implementations of the five methods defined in Chapter 4 for solving symmetric, positive definite, banded linear systems, which are designed with utility, flexibility, and portability as well as efficiency in mind.

A primary objective was to make the package flexible and yet convenient to use with a minimum of knowledge about the internal workings of the programs. To this end, BESS incorporates the following characteristics, which we discuss for the remainder of this section:

1. Argument lists are as short as possible.

2. There is some error checking *and reporting aimed at avoiding* the misuse of the codes.

3. The input file that the user must supply *is in a* straightforward format, independent of the method and the strip or block size to be used (with the sole exception of the BM method).

4. All I/O operations, such as opening and closing files, reading and writing records, and backspacing, *are isolated in* a module of simple I/O subroutines. They can easily be replaced or adapted to take advantage of a machine's specific I/O characteristics without modifying or understanding the methods that use them.

5. Routines are provided not only for computing the factorization, forward-solve, and back-solve with a first right-hand side, but also for succeeding forward- and back-solves using the same factorization with different right-hand sides.

The user must supply the following information to each subroutine. Scalar arguments are: N and M, the dimension and bandwidth of the system; L, the strip or block size; and IA and IU, the unit numbers of disk files for A and U, respectively. The user must also supply two arrays: X, initially containing the right-hand-side b to be overwritten by the solution x; and A, a work area for carrying out the factorization. Finally, the user must supply the declared length of A in the scalar argument LNGA and an error code variable IERR.

LNGA and IERR allow each method to do some error checking by verifying that there is enough space to store the principal and subordinate elements required by that method. If not, then the subroutine immediately returns with the error indicated by the value of

IERR, and the value of LNGA is set to the minimum dimension of A required to execute successfully. If a routine encounters a division by zero or a negative square root, then it also returns with an error code in IERR.

The codes that implement the SR, ST, SS, and BC methods all accept a universal input file format. This file must be a FORTRAN sequential binary (unformatted) file with each record consisting of the elements of a single column of the upper band of A, in order of increasing row index. The first M short columns must be padded with initial zeroes, so that each record contains M+1 elements with the diagonal element being the last element. Each routine reads this file and assembles the principal strips or block-columns as it requires them. The BM method cannot use this form of input file, since it never *stores a full column* in primary memory at one time, so it must be supplied with an input file containing A stored by blocks. The records of the file containing U vary with the method and with the strip or block size, but the user never has to directly manipulate this file.

Perhaps the main drawback of secondary storage methods is the possible lack of portability involved with programs that use I/O. The BESS package uses the FORTRAN-20 binary sequential and random access I/O constructs [10]. These are constructs that are standard to many FORTRAN compilers and run-time systems. However, the characteristics of I/O vary so greatly between hardware environments and operating systems that

it may be necessary or desirable to tailor the manner in which I/O is carried out. This is the motivation behind the fourth feature of BESS. All I/O is carried out by simple subroutines, which can be replaced by any other type of I/O that might be available on a specific machine without modifying the numerical portions of the package. The DEC-System 2060 implementation of these routines are described in the next [illegible]tion.

Another option in the use of BESS is offered by the module of I/O routines. In some applications, it may be possible to generate the columns of A independently and individually. A simple example is the model problem: the five-point finite-difference operator for Poisson's equation applied over a square domain. This problem yields a symmetric positive definite linear system in which a given column of A is identical to one of four simple forms, which can be determined given the parameters of the problem and the column index. In such a case, it may be desirable to assemble the columns directly in memory when they are first needed. This can be carried out with BESS by writing an assembly subroutine to replace the the initial input subroutine. This would eliminate the I/O needed to create and read the input file.

Finally, BESS includes subroutines for performing a forward-back-solve using values of U as previously stored by strips or blocks in secondary storage. The file containing U is read one record at a time in forward order to compute the forward-solve, then in reverse order for the back-solve. There are three versions for the three types of files

created by the factorization methods: a sequential file of strips created by SR or ST; a random access file of strips created by SS; and a random access file of blocks *created* by *BC* or *BM*. Naturally, the strip or block size L must be the same as was used in the factorization routine that *created the file*.

In the next section, we report on timings that show sequential or random access I/O to be fastest when reading the records of a file in forward order. This characteristic is likely to be true for most systems. Therefore, if numerous forward-back-solves are to be computed, it may be desirable to create a second file containing the records of U in reverse order to be used for the back-solve. This could speed up the I/O-dominated forward-back-solve by a substantial amount. *This is not* implemented in the BESS package, but would require only a few changes to the existing codes.

7.2 Characteristics of I/O Performance

In Chapter 5, we introduced a linear model *for the time* required by an I/O event and used the model to predict I/O costs of the methods of Chapter 4. In this section, we *describe the actual* performance of FORTRAN I/O subroutines on the DEC-System 2060 with an RP-06 disk drive. We show examples of the BESS I/O *subroutines in Figure* 7-1.

The timings contained in Tables 7-1 and 7-2 demonstrate the


```
C
      SUBROUTINE SOUT(A,LA,IFIL)
C Sequential output from A, length LA, to file IFIL.
      DIMENSION A(LA)
      WRITE (IFIL) A
      RETURN
      END
C
      SUBROUTINE RIN(A,LA,IREC,IFIL)
C Random access input into A, length LA, from record IREC of file IFIL.
      DIMENSION A(LA)
      READ (IFIL#IREC) A
      RETURN
      END
C
```

Figure 7-1: Sample BESS I/O Subroutines

performance of these FORTRAN sequential and random access I/O subroutines. The battery of tests shows some of the effects of how the I/O is carried out by the FORTRAN I/O system. For several record lengths, we timed sequential and random access transfers in three ways. First, we read a single record, which requires at least one transfer from the disk. Second, we sequentially read the records of an entire file to determine the average transfer time. The latter times are less because the system transfers records of fixed size from the disk into a buffer area and transfers them to the user's program as they are requested. Thus, some reads may not actually require transfers from the disk. In the third test for sequential I/O, we read a file in backwards order, as required in the back-solve. In the third test for random access I/O, we read records in random order to determine the average transfer time. For each case, we show the CPU time and wall time

required for the transfers and the coefficients of σ and τ for the line that best fits the wall time values.

We plot the wall time versus record size for each of these cases in Figure 7-2. These times show several departures from the simple linear I/O model we used in Chapter 5. For example, there is a considerable difference between the performance of sequential and random access I/O. For the single record times, the line which best fits the data (in a least-squares sense) gives values of σ=21 ms. and τ=.02 ms. for sequential I/O, and σ=41 ms. and τ=.04 ms. for random access. As would be expected, the time for an individual transfer in both modes is greater that the average time for many such transfers in sequential order. Furthermore, the sequential average time is faster than for transfers in backwards or random order. Random access I/O of records in arbitrary order yields values of σ=17 ms. and τ=.05, while the remaining modes of I/O have σ values which are effectively zero.

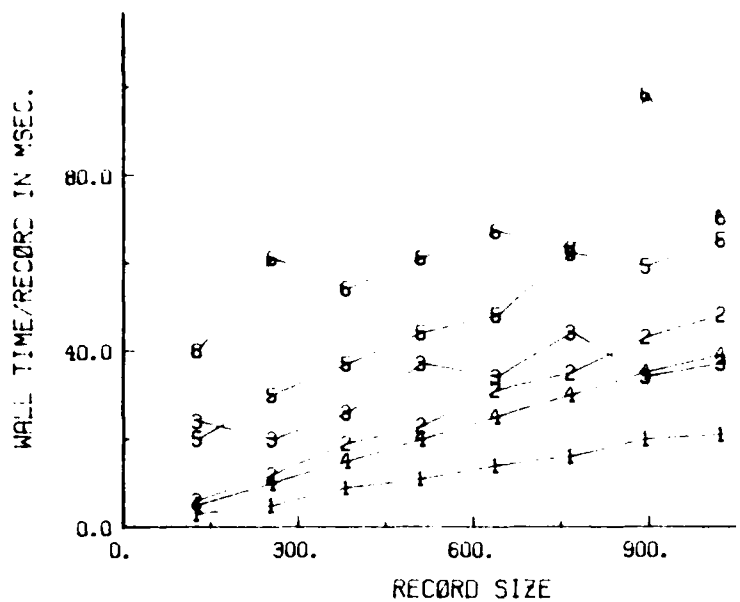
Since sequential I/O is substantially faster than random access, it is used whenever possible in implementing the methods: for all I/O in the SR and ST methods, and for the initial input of strips or blocks of A in the SS, BC and BM methods.

Record Size (Words)	Sequential I/O times in ms. Single record CPU	Single record Wall	Per record, forward seq. CPU	Per record, forward seq. Wall	Per record, backward seq. CPU	Per record, backward seq. Wall
128	7	24	2	3	4	6
256	8	20	3	5	6	12
384	9	26	4	9	8	19
512	10	37	5	11	10	23
640	10	34	6	14	11	31
768	11	44	8	16	13	35
896	13	34	9	20	15	43
1024	13	37	10	21	18	48
Best fit		ms.		ms.		ms.
σ		21.		.32		.071
τ		.02		.021		.047

Table 7-1: Timings of Sequential I/O

Record Size (Words)	Random Access I/O times in ms. Single record CPU	Single record Wall	Per record, forward seq. CPU	Per record, forward seq. Wall	Per record, random order CPU	Per record, random order Wall
128	8	40	2	5	7	20
256	8	61	3	10	9	30
384	9	54	4	15	10	37
512	12	61	6	20	11	44
640	11	67	7	25	12	48
768	12	63	8	30	14	62
896	16	98	9	35	15	59
1024	16	70	10	39	16	65
Best fit		ms.		ms.		ms.
σ		41		.25		17
τ		.04		.038		.05

Table 7-2: Timings of Random Access I/O



1 SEQUENTIAL, 60 RECØRDS FØRWARDS
2 SEQUENTIAL, 60 RECØRDS BACKWARDS
3 SEQUENTIAL, SINGLE RECØRD
4 RANDØM ACCESS, 60 RECØRDS FØRWARDS
5 RANDØM ACCESS, 60 RECØRDS RANDØMLY
6 RANDØM ACCESS, SINGLE RECØRD

Figure 7-2: Timings of Sequential and Random Access I/O

7.3 Performance of BESS on Various Problems

We now present results of time trials carried out on the BESS subroutines and, for comparison, on similar codes which store the entire matrix in primary memory. The timings were carried out on a DEC System 2060 with the TOP 20 operating system and the optimized code of the FORTRAN-20 compiler. The timings were made on a stand alone basis (i.e., a single user) so that the elapsed times would indicate the extent and effect of I/O on the time of solution without the effects of timesharing.

The virtual memory address space of this machine happened to be fairly small, and less than the amount of physical memory. This meant that the amount of paging involved in carrying out the stand-alone solution in primary memory was negligible, since there was no competition for the available primary memory. This also limited the size of a problem that could be solved in primary memory. Thus, we solved systems in which $N=M^{3/2}$ with an upper limit of N=1000, so that we could solve problems with large bandwidths.

Table 7-3 summarizes the time required by the various methods for solving a symmetric positive definite system of bandwidth 100. The timings are expressed in milliseconds per column. Thus, the value of N does not affect the primary memory requirements or relative performance

of the BESS subroutines, although it does affect the size of a system that can be solved in primary memory. We include the time for solving the system in primary memory when A is initially read from and U is written to disk in one record. We also include a timing of the ST method implemented in a row-oriented outer-product form. The outer-product algorithm is *more efficient for this method only*, but we nevertheless use the inner-product algorithm in BESS for the sake of having a universal form for input files.

The wall times and the CPU times are plotted against primary memory usage in Figures 7-3 and 7-4. With one notable exception, the relative performance of the secondary storage methods is similar to that predicted by the I/O functions of Chapter 5. The exception is that the BC method performed better than expected in comparison to the BM method. This can be explained by the observation in the previous section that random access I/O takes longer when the records being read are not in sequential order. The BM method reads two subordinate blocks at a time from separate columns, so successive reads are never in sequential order. However, the BC method retains the first subordinate set in primary memory with the principal block-column. The blocks containing the second subordinate set are then read from their block-columns in sequential order. This higher degree of locality in BC's random access I/O operations, which is not taken into account in the model used to predict I/O costs, explains why its performance is better than expected. Since the BM subroutine is outperformed by BC, and it cannot use the

same universal form of input file as the other BESS subroutines, we do not recommend it as a practical method under most situations.

The CPU timings show the extent of computational overhead of the methods due to I/O, extra loop overhead in reordering the operations, and subroutine calls for the block operators. In the ST method, we see the high CPU cost associated with the shifting of elements in the subordinate triangle, which Table 7-3 shows is much less in the outer-product form of ST. The extend of CPU overhead would vary between compilers.

Table 7-4 contains the results of several trials comparing the fragmentation of the methods due to bad choices of strip or block size. The effects of fragmentation are greatest in the block methods, where increasing the block size from 33 to 34 reduces the wall time by between 10 and 15 percent. Furthermore, the BC method uses less memory with the larger block size because there are fewer blocks per block column.

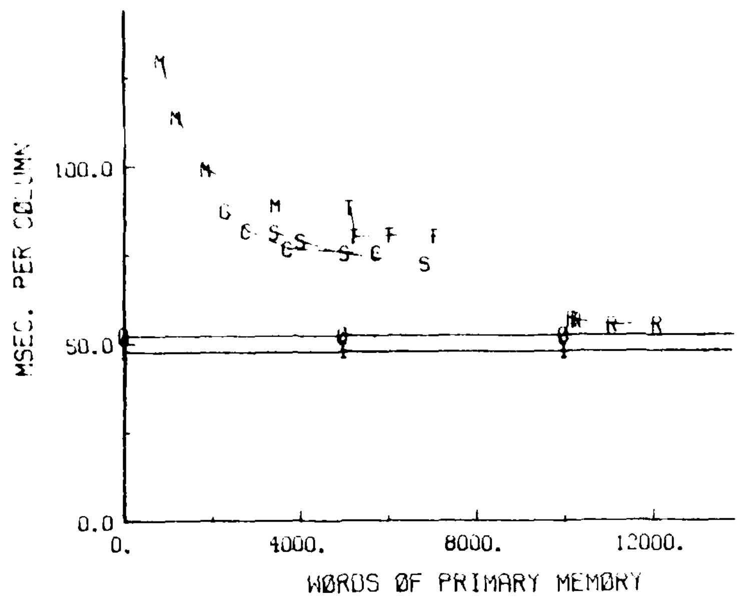
Method	Primary Memory	Record Size	Memory Used for A	Ms. per Column CPU / Wall
			(N=1000)	
In-Core	N(M+1)		101000	44.9 / 47.7
	Including I/O of A and U:			47.9 / 53.8
SR	(K+M)(M+1)	K = 1	10201	49.0 / 56.6
		K = 2	10302	47.9 / 56.1
		K = 10	11110	46.0 / 55.0
		K = 20	12120	46.0 / 55.0
ST	(K+M/2)(M+1)	K = 1	5151	71.3 / 88.0
		K = 2	5252	69.8 / 79.8
		K = 10	6060	68.4 / 80.2
		K = 20	7070	68.1 / 79.8
(Outer-Product Form:		K = 10	6060	58.5 / 67.8)
SS	2K(M+1)	K = 17	3434	55.3 / 81.2
		K = 20	4040	53.1 / 78.7
		K = 25	5050	52.1 / 75.0
		K = 34	6868	51.4 / 72.0
BC	$(2+M)L^2$	L = 17	2312	66.3 / 87.4
		L = 20	2800	59.9 / 81.8
		L = 25	3750	57.4 / 76.5
		L = 34	5780	55.8 / 75.1
BM	$3L^2$	L = 17	867	76.2 / 130.
		L = 20	1200	67.5 / 114.
		L = 25	1875	61.9 / 99.1
		L = 34	3468	58.2 / 89.0

Table 7-3: Timings and Storage of BESS Methods, M=100

Method	Record Size	Memory Used for A	Ms. per Column CPU / Total
SS	K = 25	5050	52.1 / 75.0
	K = 33	6666	55.0 / 75.5
	K = 34	6868	51.4 / 72.0
BC	L = 25	3750	57.4 / 76.5
	L = 33	6534	58.7 / 85.4
	L = 34	5780	55.8 / 75.1
BM	L = 25	1875	61.9 / 99.1
	L = 33	3267	67.2 / 104.
	L = 34	3468	58.2 / 89.0

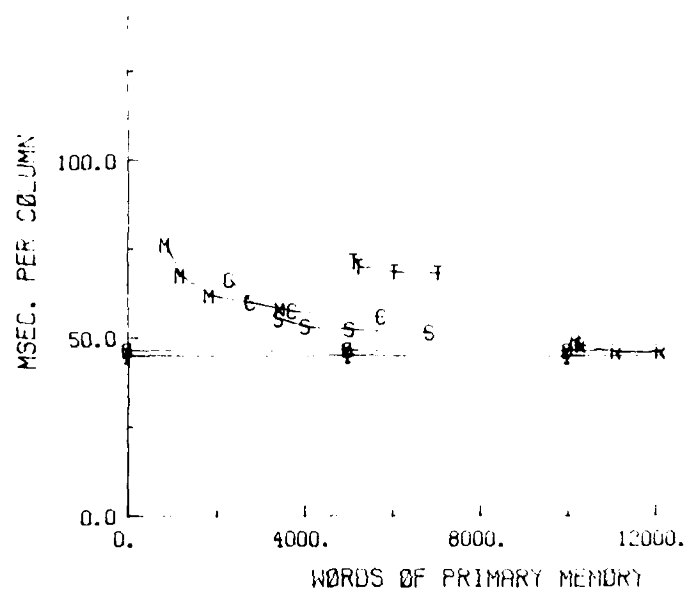
Table 7-4: Fragmentation in Bad Record Sizes, M=100

In Table 7-5 we present the results of trials carried out with several bandwidths. In this case, we constrained strip and block sizes so that $\tilde{M}$ and $\bar{M}$ were constant at various levels, as in Chapter 5. As a result, all methods use $O(M^2)$ primary memory. In Figure 7-5 we plot the wall times against bandwidth for methods that perform best within given ranges of primary memory usage. The figure shows that the *relative* performance of methods is independent of bandwidth. Furthermore, as the bandwidth grows, the *additional costs* of secondary storage methods are being *dominated* by the in-core solution time. Finally, Table 7-6 lists timings of the forward-back-solve routines for solving additional right-hand sides.



I IN-MEMØRY (NM WØRDS ØF STØRAGE)
Ø IN-MEMØRY WITH I/Ø ØF A AND U
R STRIP-RECTANGLE
T STRIP-TRIANGLE
S STRIP-STRIP
C BLØCK-CØLUMN
M BLØCK-MINIMUM

Figure 7-3: Wall Time vs. Primary Memory, M-100

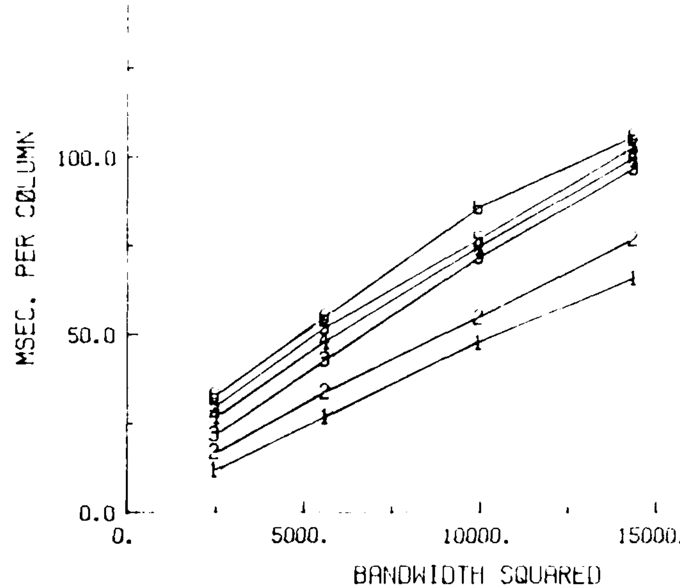


I IN-MEMØRY (NM WØRDS ØF STØRAGE)
Ø IN-MEMØRY WITH I/Ø ØF A AND U
R STRIP-RECTANGLE
T STRIP-TRIANGLE
S STRIP-STRIP
C BLØCK-CØLUMN
M BLØCK-MINIMUM

Figure 7-4: CPU Time vs. Primary Memory, M-100

Method, Record Size	Approx Memory Used for A	Ms. per Column (CPU / Wall) for Bandwidths of 50	75	100	120
In Memory	NM	11 / 12	25 / 27	45 / 48	62 / 66
SR, K-M/5	1.2 M^2	14 / 17	27 / 34	46 / 55	65 / 77
ST, K-M/5	.70 M^2	19 / 24	40 / 50	68 / 80	96 / 111
SS, K-M/3	.67 M^2	16 / 22	31 / [illegible]	51 / 72	72 / 97
K-M/4	.50 M^2	16 / 24	31 / 46	52 / 75	74 / 100
K-M/5	.40 M^2	18 / 25	32 / 49	53 / 79	77 / 106
BC, L-M/3	.56 M^2	19 / 27	34 / 48	56 / 75	76 / 100
L-M/4	.38 M^2	21 / 30	37 / 52	57 / 77	78 / 103
L-M/5	.28 M^2	25 / 33	39 / 55	60 / 82	81 / 106
BM, L-M/3	.33 M^2	20 / 34	36 / 58	58 / 89	79 / 111
L-M/4	.19 M^2	25 / 41	41 / 67	62 / 99	84 / 123

Table 7-5: BESS Timings for Various Bandwidths



1 IN-MEMØRY, IN NM MEMORY
2 STRIP-RECTANGLE, IN 1.2 M**2 MEMORY
3 STRIP-STRIP, IN .67 M**2 MEMORY
4 BLØCK-CØLUMN, IN .56 M**2 MEMORY
5 BLØCK-CØLUMN, IN .38 M**2 MEMORY
6 BLØCK-CØLUMN, IN .28 M**2 MEMORY

Figure 7-5: Wall Times of BESS Methods vs. Bandwidth

Method, Record Size	Approx. Memory Used for A	Ms. per Column (CPU / Wall) Bandwidth 50	75	100	120
In Memory	NM	.6/.6	1.1/1.2	1.5/1.6	1.8/1.9
By Strips, K=M/5	.20 M^2	2.3/4.2	3.1/6.6	4.1/9.5	4.7/11.
By Blocks, L=M/4	.06 M^2	3.1/6.2	4.2/8.8	5.1/12.	5.9/15.

Table 7-6: Timings of Forward- and Back-Solve Routines

7.4 Experimental Memory Occupancy Costs

In Chapter 5, we derived asymptotic occupancy costs for the various methods compared with band and sparse elimination in primary memory. We now use the timings and primary requirements from Table 7-5 to compute experimental memory occupancy costs for the methods. In Figure 7-6, we plot memory occupancy versus primary memory usage for several bandwidths, where each point represents a specific choice of method and block or strip size.

This figure supports the same conclusion as the analysis of previous chapters: the less primary memory used, the smaller the occupancy costs. Whether smaller occupancy costs are offset by larger

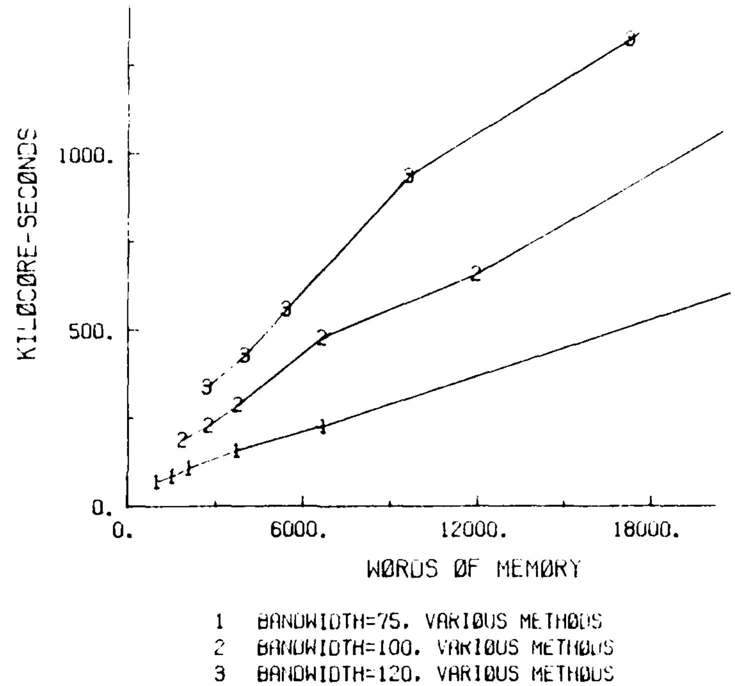


Figure 7-6: Memory Occupancy vs. Primary Memory Usage

I/O costs *depends on the relative* charges associated with each cost at a given installation, or on the priorities of the user.

These experimental results corroborate the mathematical analysis of costs in Chapter 5. They indicate that secondary storage methods can dramatically decrease the primary memory requirement of solving symmetric, positive definite banded linear systems, and in turn the memory occupancy costs, without prohibitive increases in turn-around time. There is an alternative scheme called minimal-storage band elimination [6] for reducing the primary memory requirement to about M^2. This scheme throws away and later recomputes elements in a recursive scheme that increases the work by less than 100 percent for the model problem. We see here that the same reduction in primary memory is achieved by the SR method with an almost negligible increase in turn-around time. Much larger reductions are possible by other methods within a 50 percent increase in run time. Not only would the BESS codes be expected to run faster than minimal-storage band elimination on many machines, but I/O time is usually less expensive than a similar amount of CPU time.

Obviously, the performance reported in this chapter is highly dependent upon the computer, its operating system, and the secondary storage device. But these results were achieved using a straightforward, high-level implementation of I/O, little attention to the location or contiguity of data files on the disk, and hardware which

is not unusually well suited to this purpose. We have shown that secondary storage methods offer a better trade-off between storage and I/O than paging systems, since they exhibit good performance over the entire range of primary memory usage. The possibility of overlapped I/O would make the methods even more attractive, as investigated in Chapter 6. In theory and in practice, secondary storage methods are an efficient means for solving large, symmetric, positive definite, banded linear systems within limited amounts of memory.

Bibliography

[1] W. A. Abu-Sufah.
Improving the Performance of Virtual Memory Computers.
PhD thesis, Computer Science Department, University of Illinois at Champaign-Urbana, 1978.

[2] P. Alexander.
The array processor as an intelligent simulation co-processor.
In Proceedings of the 1979 Summer Computer Simulation Conference. Society for Computer Simulation, 1977.

[3] D. A. Calahan, P. G. Buning and W. N. Joy.
Vectorized General Sparsity Algorithms with Backing Store.
Technical Report 96, Systems Engineering Laboratory, University of Michigan, 1977.

[4] E. Cuthill and J. McKee.
Reducing the bandwidth of sparse symmetric matrices.
Proceedings, ACM National Conference, 1969.

[5] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart.
LINPACK User's Guide.
SIAM, Philadelphia, 1979.

[6] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Minimal storage band elimination.
In A. H. Sameh and D. Kuck, editor, Proceedings of the Symposium on Sparse Matrix Computations, pages 273-286. University of Illinois at Champaigne-Urbana, April, 1977.

[7] K. Fong and L. Jordan.
Some Linear Algebraic Algorithms and Their Performance on CRAY-1.
Technical Report 6774, Los Alamos Scientific Laboratory, 1977.

[8] George Forsythe and Cleve B. Moler.
Computer Solution of Linear Algebraic Systems.
Prentice-Hall, 1967.

[9] FORTRAN Extended Version 4 Reference Manual.
Control Data Corporation, 1977.

[10] FORTRAN Reference Manual, FORTRAN-20.
Digital Equipment Corporation, 1977.

[11] Dieter Fuss.
Postscript on overlapped I/O.
The Buffer 4(2):10-11, 1980.
Published by National MFE Computer Center, Livermore Laboratory.

[12] Alan George.
Nested dissection of a regular finite element mesh.
SIAM J. Numer. Anal. 10(2):345-363, 1973.

[13] Alan George.
Direct methods for solving large sparse systems: part two.
SIAM News 13(4), Aug., 1980.

[14] E. M. Hill.
Computer Solution of Large Dense Linear Problems.
PhD thesis, Department of Computer Science, University of Maryland, 1977.

[15] A. C. Hindmarsh.
DISBAND: Disk-Contained Symmetric Band Matrix Solver.
Technical Report UCID-30065, Lawrence Livermore Laboratory, 1973.

[16] Computer Subroutine Libraries in Mathematics and Statistics.
International Mathematical and Statistical Libraries, Inc., 1977.

[17] B. M. Irons.
A frontal solution program for finite elements.
International Journal for Numerical Methods in Engineering 2:5-32, 1970.

[18] A. Jennings.
A compact storage scheme for the solution of symmetric linear simultaneous equations.
Computer Journal 9:281-285, 1966.

[19] H. A. Kamel and M. W. McCabe.
Direct numerical solution of large sets of simultaneous equations.
Computers and Structures 9:113-123, 1978.

[20] John C. Knight, William G. Poole Jr., and Robert G. Voigt.
System Balance Analysis for Vector Computers.
Technical Report 75-6, ICASE, Langley Research Center, 1975.

[21] H. T. Kung and C. E. Leiserson.
Systolic arrays (for VLSI).
In Sparse Matrix Proceedings 1978, pages 256-282. SIAM, 1979.

[22] R. S. Martin and J. H. Wilkinson.
Symmetric decomposition of positive definite band matrices.
Numerische Mathematik 7:355-361, 1965.

[23] A. C. McKellar and E. G. Coffman Jr.
Organizing matrices and matrix operations for paged memory systems.
CACM 12(3):153-165, 1969.

[24] C. B. Moler.
Matrix computation with FORTRAN and paging.
CACM 15(5):268-270, 1972.

[25] Digambar P. Mondkar and Graham H. Powell.
Large capacity equation solver for structural analysis.
Computers and Structures 4:699-728, 1974.

[26] D. A. Orbits and D. A. Calahan.
Data Flow Considerations in Implementing a Full Matrix Solver with Backing Store on the CRAY-1.
Technical Report 98, Systems Engineering Laboratory, University of Michigan, 1977.

[27] B. N. Parlett and Y. Wang.
The influence of the compiler on the cost of mathematical software.
ACM Transactions on Mathematical Software 1(1):35-46, 1975.

[28] John M. Pavkovich.
The Solution of Large Systems of Algebraic Equations.
Technical Report 33, Computer Science Department, Stanford University, 1963.

[29] Suresh R. Phansalkar.
Solution of large systems of linear simultaneous equations by inverse decomposition.
Computers and Structures 5:131-144, 1975.

[30] J. K. Reid.
Two FORTRAN Subroutines for Direct Solution of Linear Equations Whose Matrix is Sparse, Symmetric Positive-Definite
1972.

[31] J. R. Rice.
ELLPACK, A research tool for elliptical partial differential equations software.
In J. R. Rice, editor, Mathematical Software III, . Academic Press, 1977.

[32] L. D. Rogers.
Optimal Paging Strategies and Stability Considerations for Solving Large Linear Systems.
PhD thesis, Computer Science Department, University of Waterloo, 1973.

[33] William T. Segui.
Computer programs for the solution of systems of linear algebraic equations.
International Journal for Numerical Methods in Engineering 7:479-490, 1973.

[34] Andrew H. Sherman.
On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations.
PhD thesis, Computer Science Department, Yale University, 1975.

[35] G. D. Smith.
Numerical Solution of Partial Differential Equations.
Oxford University Press, 1965.

[36] John Stewart.
Overlapped I/O on the CRAY-1.
The Buffer 4(1):8-11, 1980.
Published by National MFE Computer Center, Livermore Laboratory.

[37] K. S. Trevidi.
Prepaging and Application to Structured Array Problems.
PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaigne, 1974.

[38] James H. Wilkinson.
The Algebraic Eigenvalue Problem.
Oxford University Press, 1965.

[39] Edward L. Wilson, Klaus-Jurgen Bathe and William P. Doherty.
Direct solution of large systems of linear equations.
Computers and Structures 4:363-372, 1974.

[40] David M. Young.
<u>Iterative Solution of Large Linear Systems</u>.
Academic Press, 1971.

DATE

FILMED